Functional Web Applications

Implementation and Use of Client Side Interpreters

Jan Martin Jansen

ISBN: 978-90-9025436-4
NUR-code: 988

Typeset with LaTeX2ε
Cover design by P.J. de Vries, Bureau Multi Media NLDA
Printed by Ipskamp Print Partners

This research was supported by the Netherlands Defence Academy (NLDA) and the Centre for Automation of Mission-Critical Systems (CAMS-Force Vision)

Functional Web Applications

Implementation and Use of Client Side Interpreters

Een wetenschappelijke proeve op het gebied van de
Natuurwetenschappen, Wiskunde en Informatica

Proefschrift

ter verkrijging van de graad van doctor
aan de Radboud Universiteit Nijmegen
op gezag van de rector magnificus prof. mr. S.C.J.J. Kortmann
volgens besluit van het college van decanen
in het openbaar te verdedigen op donderdag 8 juli 2010
om 13:30 uur precies

door

Johannis Martinus Jansen

geboren op 14 maart 1960 te Nieuwe Niedorp

Promotor:
Prof. dr. dr. hc. ir. M.J. Plasmeijer

Copromotor:
dr. P.W.M. Koopman

Manuscript commissie:
Prof. dr. F.W. Vaandrager
Prof. dr. C. Runciman University of York
Prof. dr. S.D. Swierstra Universiteit van Utrecht

Contents

Chapter 1

Introduction

This thesis is about the use of functional programming languages for the implementation of Internet applications and, in particular, the use of interpreters for client-side processing. This chapter gives an overview of the scope and the topics of the research. After a discussion about what makes Internet application development difficult and how modern functional programming can help to ease this development, a detailed description of the contents of this thesis and the contributions of the author is given.

1.1 Internet Applications

During the last decade the Internet has become the prominent platform for the deployment of computer applications and web-browsers are an important interface for a large class of computer applications, such as e-mail applications, on-line shops and banking applications. Furthermore, they are used as the default communication interface between customers and companies like governmental institutions, insurance companies, etc.

An important advantage of using web-browsers to interface with applications is that they do not require installation of application related software on a computer to use them. It is even possible to run the same web application on a large number of different platforms and operating systems, including PDA's, smart phones, etc.

Despite this popularity and convenience for the user, for a software engineer the development of web application is a difficult job. There are several reasons for this. First, web browsers were originally designed for browsing through HyperText documents (displaying text and links between pages). Although the use of web browsers has changed significantly, their design is adapted only just enough to accommodate the new requirements. This complicates the development of desktop like applications which make use of a web browser for their interface instead. Second, Internet applications follow the client-server paradigm and consequently have a more complex structure than desktop applications. Applications have hardly any control over clients. Clients have usually very limited privileges on the machine executing them. The client can become active (again) after an arbitrary delay. The browser adds behavior to the client (e.g. by back and forward buttons and cloning of pages). The

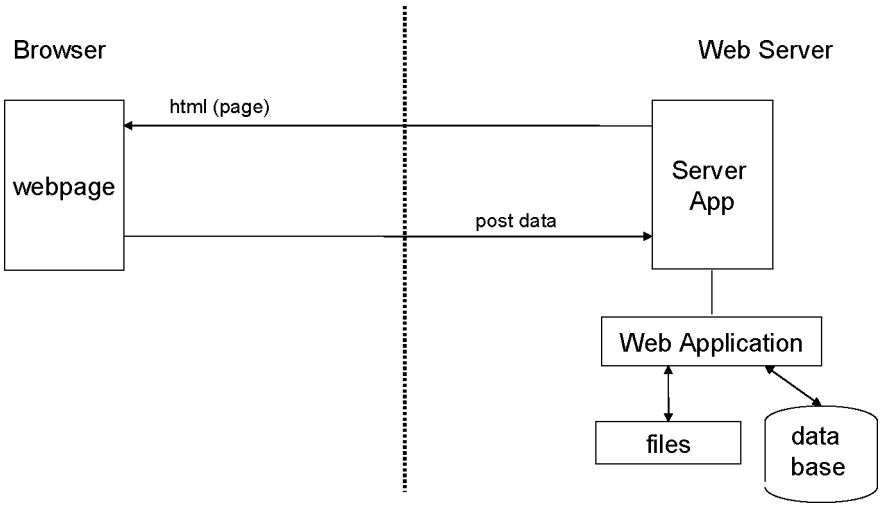


Figure 1.1: The traditional architecture of an Internet Application

quality, delay and bandwidth of the connection between client and server varies enormously.

Fig. 1.1 shows the typical architecture of a traditional Internet application. The client browser (left from the dashed line) only displays the Html generated by the (web)application running on the web server (right from the dashed line). The user can fill in web forms that are sent to the server and processed there (as post data). As a result the server produces a new web page that is displayed at the client side. The (web)application can read/write information from/to data bases or files residing at the server side. These data bases and files are used to maintain information like the user login name or the purchase the user made. An important issue that has to be dealt with is maintaining the state of a transaction. The application has to keep track of this state. As said before, a complicating factor is that the user can move away from a web-page at any moment and come back to the web page at a later moment (using the back and forward buttons) or clone web-pages and continue a transaction in a page that is in a different state. As a consequence the state of an application must also take the currently used web-page into consideration.

1.1.1 Client-side Processing using the Ajax Paradigm

In the classical setting the web server processes a web form filled in by the user and produces a new Html page. A drawback of this approach is that the system as a whole becomes less responsive because large amounts of data need to be (re)transmitted after each user action. To overcome this, local processing at the client side is necessary, e.g. for checking the format of user input in text fields. But it is no valid option to execute native code that is part of the Internet application, at the client side for two reasons. First, the client platform is unknown beforehand. So one should have code available for all possible execution platforms. Second, executing native code

at the client side causes great security risks. Malicious client-side code may easily harm the client computer. Therefore, interpreters for programming languages are used for client-side processing. In contrast to a compiler an interpreter does not translate (compiles) a program to executable code, but directly executes (interprets) this code. This has a number of advantages. First, the interpreted program cannot harm the computer it is running on because it runs within the interpreter environment (also called a sandbox) and the interpreter takes care that no harm can be done. Second, if an interpreter for a certain platform is available, it can run arbitrary programs written in the interpreted language. Therefore, it is not necessary any more to compile the programs for each platform separately. The price to pay is that interpreted programs run slower than compiled programs. Furthermore, the interpreter must be made available for all platforms one wants to run client-side code. An interpreter platform especially made for web-browsers is JavaScript which is integrated in all modern web browsers. JavaScript programs can be integrated within Html descriptions of web-pages and such programs have access to the content of the web page (can process user input in web forms and make updates to the page). JavaScript offers the web-programmer a light-weight platform for doing client-side processing for which no information residing at the server side is needed. Typical use cases for JavaScript are to perform sanity checks on user input or to adapt the layout of information in a web page. To further enhance the performance of web applications it is even possible to make asynchronous requests to the server from within JavaScript. The results of these requests can be used to update the web page. This technique is known under the name Ajax (Asynchronous JavaScript And XML) [Gar05]. Here XML is often (but not necessarily) used for encoding data in a Ajax request. Using Ajax in general leads to less data being sent back and forth between server and client, which can enhance the responsiveness of web applications considerably. An important class of web applications that work in this way are the so-called Web 2.0 applications. These are applications that support interactive information sharing, user editing and collaboration. Examples are social networks, wikis, blogs and mash-ups. Fig. 1.2 shows the architecture of web applications using the Ajax paradigm. Google extensively uses this technique in applications like GMail, GoogleDocs and GoogleMaps to speed up their performance and make them more interactive.

1.1.2 Client-side Processing using Java

An alternative client-side processing platform is Java. Java combines an interpreted and compiled approach in a single formalism. Java programs are executed using a virtual machine. They are compiled to Java byte code instructions that can run on (are interpreted by) the Java virtual machine (JVM). Almost all popular web-browsers have a JVM plug-in available. As a consequence a web developer only has to write and compile a single Java application that can then run on a large number of platforms. Much effort has been spent to make the JVM virtual machine as fast as possible. An important technique to achieve this is the use of a Just-In-Time (JIT) compiler. When executing byte code instructions they are translated (compiled)

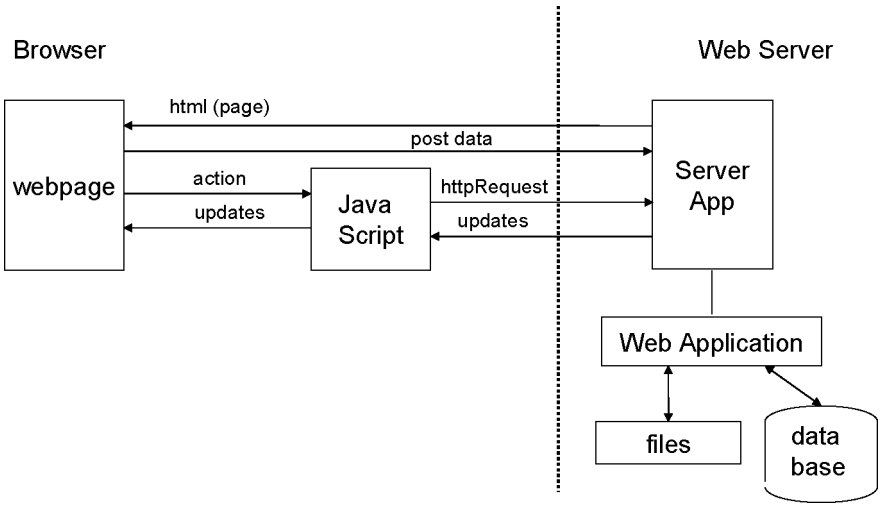


Figure 1.2: The architecture of an Internet Application using the Ajax paradigm

on-the-fly to native code for the platform the virtual machine is running on. When the code is executed a second time the native code can be used. This has resulted in Java applications that run at speeds comparable to their machine coded equivalents. Because native code is generated in a controlled environment, one can enforce the safety of a purely interpreted implementation. Java virtual machines including JIT compilation are available for a large variety of platforms.

Using Java at the client side of web-application does not offer the ease of use that JavaScript offers. JavaScript is an integral part of the web-browser (does not require the loading of a plug-in) and has easier/better access to the contents of web-pages. Nowadays also JavaScript uses JIT compilation techniques to speed-up its processing. Nevertheless, through its portability and speed Java is still an attractive option for fast processing at the client side of web applications.

1.1.3 The Complexity of Internet Application Development

When developing web applications with client-side processing, the developer has to deal with several formalisms: server-side programming languages like Java, C(++) or PHP; server-side data-base access languages like SQL; client-side programming languages like JavaScript, Java or VBScript using Ajax calls for information exchange with the server application; Html for the contents of web pages. Program parts made in these different formalisms have to collaborate smoothly to achieve the desired result. For example, data conversion between the formats used at client and server side is necessary. This all complicates software development for the Internet considerably.

Many frameworks for developing web-applications using a variety of programming platforms are available. Basically three approaches are used. In the first one a programming environment (Integrated Development Environment or IDE) automatically generates a framework that must be complemented with user written code

using the different programming formalisms for client and server code. In the second approach a Domain Specific Language (DSL) is designed for describing web applications. A specially designed compiler is used to compile these descriptions to (a collection of) programs in the server and client-side programming formalisms. All extra code that is necessary for communication between server and client, including data conversion code is generated too. In the third approach a graphical formalism is used for describing the generic structure of a web-application. From this generic structure application framework programs are generated in server and client-side formalisms that often must be complemented with user written code. All these approaches have their pro's and con's. Using the DSL approach one has to deal with only one formalism, which simplifies the development and maintenance of the software. Graphical formalisms allow for the rapid development of applications, without the need for the developer to learn complicated formalisms. A disadvantage of the DSL approach is that DSL's are always restricted formalisms. They do not offer the programming power of a general purpose programming language. As a consequence the generated code often must be adapted or complemented with code that could not be expressed in the DSL. Graphical formalisms are also restricted in the constructs that can be expressed. As a consequence only a limited set of applications can be expressed, and again one has to resort to adding supplementary code in the generated formalisms.

1.2 Internet Applications and Functional Programming

In this thesis we advocate an approach which uses a general purpose functional programming language for the realisation of web applications. In this approach all web form (Html) generation and all communication between server and client is handled automatically or with an absolute minimum of explicitly written code. The reasons to use functional programming languages as implementation platform are the high expressiveness of these languages with the possibility:

- to define higher order combinators that enable a high level of compositional programming where irrelevant details can be hidden for the developer;
- to use generic (type driven) programming techniques for automatic generation and handling of web forms, interaction with data sources and server-client communication of data types.

Especially the use of generic programming techniques offers important advantages that functional languages provide above other programming formalisms. In the next subsection we say more about this. By using combinators one can extend a functional language to an embedded DSL without getting the disadvantages mentioned above. The full programming power of the host language remains available in the DSL.

There are a number of challenges one has to deal with when using (functional) programming formalisms for both server and client-side processing. First, one should

have a sufficiently fast execution platform for both sides. For the client side this means the availability of a fast interpreter. Second, it should be possible to switch execution between server and client in a smooth way, with a minimum of burden for the application developer. Both issues are research topics for this thesis.

1.2.1 Generic Programming

Generic or polytypic programming makes it possible to define functions for a whole class of algebraic data types (ADT's). Algebraic data types are constructed from primitive types (integer, booleans, etc) and previously defined ADT's using simple composition mechanisms like sums (alternatives) and products (records). A generic function is defined on the generic structure of an ADT by just specifying its result for the primitive types and by giving an inductive recipe how to compose the result for sums and products from (the results of) the types they have been constructed from. The compiler is now capable of generating instantiations of this function for concrete types from the generic version. It is always possible for a programmer to overrule the generic implementation of a function for a concrete type by giving a specific implementation for this type. Generic programming is possible in languages like Clean [PE01] and Haskell [PJ03].

1.2.2 Existing Functional Programming Approaches to Web Programming

In the functional programming community there are already several research activities that focus on tooling for the development of web applications. Examples that use functional languages at the server side and that generate complete web-applications are WASH [Thi02] and iData [PAP05]. Both approaches generate Html web forms. iData generates Html web-forms from Clean data structures and handles user updates for them automatically by using the above mentioned generic programming techniques. With iData it is possible to implement spreadsheet-like applications with relatively little effort. iData and WASH use the traditional Internet model, where all processing is done at the server side.

For realizing applications with client-side processing and Ajax interaction between client and server one can either generate client-side JavaScript from a functional programming based specification, or include a dedicated interpreter for a functional formalism as a plug-in at the client side. Curry [Han07] and Hop [SGL06, LS07] both use the first approach and generate JavaScript from specifications made in Curry and Scheme respectively. Links [CLWY06] and its extension formlets [CLWY07] is another example of the first approach. Links compiles to JavaScript for the generation of HTML pages, and SQL to communicate with a back-end database. HaskellScript [MLH99] uses the second approach by supplying a Haskell interpreter plug-in at the client side.

1.2.3 The iTask Dynamic Workflow System

With the iData tool-kit it is possible to generate web forms from data types and to handle user changes in these forms automatically. As a result only single user and rather static web applications, containing a single web form, are easy to realize. Many web-applications are more dynamic: a user has to go through a number of web forms for making a purchase; many users are involved to accomplish something. In such applications both a flow of control and a flow of information must be maintained. Implementing these issues using iData is tedious. To overcome this iTask [PAK07] was developed. The iTask system (itasks.cs.ru.nl) is a declarative domain specific workflow language embedded in Clean, enabling the creation of dynamic workflow systems. In the iTask system a workflow consists of a combination of **tasks** to be performed by humans and/or automated processes. A workflow specification made in iTask results in a complete workflow application that runs on the web. The system is based on open web-standards and can therefore be accessed by anyone who has access to Internet via a number of web services, nowadays including many mobile devices.

The iTask system is built upon a few simple concepts. The main concept is that of a typed task. A task is a unit of work to be performed by a worker or computer (or a combination of both) that produces a result of a certain *type*. The result of one task can be used as the input for subsequent tasks, and therefore these new tasks are dynamically dependent on results of previously executed task components. iTask allows for data dependent sequential and parallel execution of tasks, with information being automatically transported between tasks.

The original version of the iTask system was a pure server-side based application, using the model from Fig. 1.1. The realization of a version of iTask supporting client side processing and using the Ajax paradigm is one of the main topics of this thesis. More details about the iTask system itself can be found in chapters 5 and 6.

1.3 Scope and Contents of this Thesis

This thesis consists of seven papers, which were all published in the open literature or have been submitted for publication. The main object of study for this research is the realization of client-side processing for iTask using a functional programming formalism and an initial investigation of applications of the iTask toolkit in the domains of crisis-management and military operations. The research of this thesis can be divided into three parts:

- The first part (chapters 2, 3 and 4) investigates the realisation of efficient interpreters that can be used at the client side of web applications. For this, one can choose between either using an existing client-side processing platform like JavaScript as target language, or for adding a dedicated interpreter plug-in at the client side. We have opted for the second approach. In this part we also investigate the formalism of the interpreter and investigate whether the techniques that were used for the implementation of the interpreter can also

be used for the realisation of an efficient compiler. Papers upon which this part is based are:

  - J. Jansen, R. Plasmeijer, and P. Koopman. Functional Pearl: Comprehensive Encoding of Data Types and Algorithms in the λ-Calculus. *Journal of Functional Programming*, Submitted for publication, 2010.
  - J. Jansen, P. Koopman, and R. Plasmeijer. Efficient interpretation by transforming data types and patterns to functions. In H. Nilsson, editor, *Selected Papers of the 7th Symposium on Trends in Functional Programming, TFP'06*, volume 7, pages 73-90, Nottingham, UK, 2006. Intellect Books.
  - J. Jansen, P. Koopman, and R. Plasmeijer. From interpretation to compilation. In Z. Horvath, editor, *Proceedings of the 2nd Central European Functional Programming School, CEFP'07*, volume 5161 of *LNCS*, pages 286-301, Cluj Napoca, Romania, 23-30, June 2008. Springer-Verlag.

- The second part (chapters 5, 6 and 7) discusses how client-side processing can be integrated into the iTask system with a minimum of disruption for the application programmer. We start with giving an overview of the iTask system and discuss our experience in developing applications with it. Next the technical realisation of Ajax and client-side processing using local and client side task-tree rewriting is described. Finally, we describe how arbitrary web plug-ins can be integrated into iTask applications. Papers upon which this part is based are:

  - J. Jansen, R. Plasmeijer, P. Achten, and P. Koopman. Embedding a web-based workflow management system in a functional language. In C. Brabrand and P.-E. Moreau, editors, *Proceedings 10th Workshop on Language Descriptions Tools and Applications, LDTA'10*, pages 79-93, Paphos Cyprus, March 27-28 2010.
  - R. Plasmeijer, J. Jansen, P. Koopman, and P. Achten. Declarative Ajax and client-side evaluation of workflows using iTasks. *Proceedings of the 10th International Conference on Principles and Practice of Declarative Programming, PPDP'08*, pages 56-66, Valencia, Spain, 15-17, July 2008.
  - J. Jansen, R. Plasmeijer, and P. Koopman. iEditors: extending iTask with interactive plug-ins. In S.-B. Scholz, editor, *Selected Papers of the 20th International Symposium on the Implementation and Application of Functional Languages, IFL'08*, 2009. To appear in Springer LNCS 5836.

- The third part (chapter 8) focuses on application areas for the iTask system. The focus is on crisis-management and military operations as a first object of study, because these are demanding application areas that are hard to support by existing workflow tooling and because of the author's affiliation. The work in this section is based on the paper:

– J. Jansen, B. Lijnse, R. Plasmeijer and T. Grant. Web Based Dynamic Workflow Systems for C2 of Military Operations. *Proceedings of the International Command and Control Research and Technology Symposium (ICCRTS) 2010, Santa Monica, USA.*

This paper is an extended version of:

– J. Jansen, B. Lijnse and R. Plasmeijer. Towards dynamic workflows for crisis management. In Mark Haselkorn and Simon French, editors, *Proceedings of the 7th International Conference on Information Systems for Crisis Response and Management'10*, Seattle, WA, USA, May 2010.

Another paper by the author describing possible applications in the military domain, but not added as a chapter in this thesis is:

– J. Jansen, P. Koopman and R. Plasmeijer. Web based dynamic workflow systems and applications in the military domain. In Theo Hupkens and Herman Monsuur, editors, *Netherlands Annual Review of Military Studies - Sensors, Weapons, C4I and Operations Research*, pages 43–59, 2008.

This paper is a precursor of the two papers mentioned above.

In the next subsections a short overview of each of the above mentioned papers is given.

1.3.1 Comprehensive Encoding of Data Types and Algorithms in the λ-Calculus

In order to describe an interpreter one should first say something about the language for which it provides a semantics. The language of the interpreter described in this thesis is an intermediate language. This means that the language is not intended to be used as a programming language directly, but that programs written in other languages are translated to this language. The intermediate language uses a minimal number of concepts, which simplifies the construction of an interpreter but it is high level enough to allow for a straightforward and easy transformation of programs written in state-of-the-art languages like Clean or Haskell to it. The language is at the same level of abstraction as the Core languages used for Clean and Haskell. This language is suitable for direct interpretation and does not require transformation to another, more low level (byte-code like) formalism. The representation of Algebraic Data Structures (ADT's) is an interesting and distinguishing feature of this intermediate language. During a discussion with Dick Bruin [Brua] in 1999 we discovered that it is possible to make an elegant representation of ADT's using pure functions only. Later on, it became clear that this way of representing ADT's was not new, but earlier discovered by Scott but never officially published by him (see also [CHS72]).

This paper investigates how this representation of ADT's can be used to express algorithms in the λ-calculus. It also compares various ways to represent ADT's in

the λ-calculus. The λ-calculus is a universal functional programming language and it forms the foundation of all modern functional programming languages. Unfortunately, λ-calculus based programs are not always easy to comprehend. We show that this qualification is not correct and caused by the traditional choice for the representation of data types in the λ-calculus: the Church encoding. We show that if we use a representation of ADT's based on the representation used in the intermediate language, we can express functional programs, resembling equivalent programs written in languages like Clean or Haskell. The only drawback is that the resulting representation cannot be typed using standard Hindley-Milner type inference as it is used for Haskell and Clean. For expressing algorithms, we also use an alternative way to express recursion without the use of a fixed-point combinator. Finally, we compare the Scott and the Church encoding and show that the connecting element between them is the `fold` function.

This paper was written by the author of this thesis under supervision of the co-authors.

1.3.2 Efficient Interpretation by Transforming Data Types and Patterns to Functions

This paper forms the core of the work on interpreters. It introduces the **S**imple **A**pplication **P**rogramming **L**anguage (SapI). In SapI the Scott encoding, as described in Chapter 2, is used for representing values by functions. A SapI program consists of (pure) function definitions only. In fact, the essential difference between SapI and the formalism of chaper 2 is the use of named functions instead of anonymous λ-expressions. Only constant **let** expressions are added to enable us to express sharing and build cyclic data definitions. Due to the encoding of instances of ADT's by functions the one-and-only basic operation in SapI is function application (β-reduction). The operational effect of a function application consists of the replacement of a function call by the right-hand-side of the function definition with all parameters replaced by the corresponding arguments. This makes it possible to define an elegant and minimal interpreter for SapI based on pure graph reduction only. Graph reduction is a well known and straightforward implementation technique for lazy functional languages (see also [Tur79], [PvE93],[PJL92] and [PJ87]). By adding integers and their associated operators to SapI and its interpreter one obtains a practically usable programming language. The interpreter implements a basic lazy functional programming language efficiently. Due to its high abstraction level it is easy to compile any higher-level lazy functional language like Haskell and Clean to SapI. Due to its conciseness it is is also suited for educational purposes.

It turns out that our interpreter can easily be optimized using a few straightforward transformations, like reducing the size of the graphs in the implementation, the inlining of function calls and by making use of a simple annotation to SapI programs. When comparing our approach with several other interpreters and compilers for functional languages using a set of representative benchmarks, it turns out that our implementation has a competitive performance. It is at least twice as fast as Amanda, [Brub], Helium, [Sof], Hugs [Hug] and GHCi [GHC] and in a number

cases, e.g. for programs involving mostly higher order functions, it is even competitive with compilers like Clean and GHC. This demonstrates that for interpreters a high-level graph-reduction based implementation in general leads to better results than the use of more low level formalisms like byte code interpreters. It also shows that keeping things simple and overhead to a minimum is the way to go to obtain efficient interpreters.

The encoding of ADT's used for SapI has also been used for an FPGA implementation of graph reduction in [NR08] (more details in chapter 9).

This paper was written by the author of this thesis under supervision of the co-authors.

1.3.3 From Interpretation to Compilation

The SapI interpreter described in chapter 3 has in some cases a performance comparable to that of compilers. This made us curious about whether it is possible to apply the techniques used for the SapI interpreter in the construction of an efficient compiler. The construction of such a compiler is the focus of this paper.

Because the interpreter was implemented in C(++), this language was also chosen as the target language for the compiler. This approach is not uncommon, GHC also uses C as target language. The analysis starts with the construction of a straightforward version of a compiler that only differs from the interpreter at the following points: graph instantiation is hard coded (instead of tree traversal by a recursive function); the generic control structure of compiled functions is hard coded instead of interpreted. The actual reduction of graphs does not differ from that of the interpreter. This version is used for a benchmark comparison with the Clean and GHC compilers. For this comparison the same benchmark programs were used as for the interpreter comparison in chapter 3. It turns out that for a number of benchmark programs the SapI compiler already has similar performance, while for other benchmark programs the performance is much worse (3-30 times slower). A detailed analysis of the latter benchmark programs reveals that they often make heavy use of purely numeric functions, tail recursive functions or a combination of them. We therefore added tail recursion detection and detection of numeric functions and (sub)expressions to the compiler. In the code generation phase we generate a loop for tail recursive calls. In the loop memory cells are reused. This saves allocation and garbage collection of cells. In the best case this led to a speed-up of a factor of 7. Purely numeric functions and (sub)expressions are replaced by their C++ equivalents. The speed-up obtained with this optimization can be a factor of 40 for purely numeric functions. These optimizations are rather straightforward to implement and result in an acceptable performance in almost all cases (less than 2-4 slower than GHC -O and Clean). This study demonstrates that with relatively little effort an excellent performance can be obtained for a large numbers of cases (80-20 rule). Unfortunately, much more effort has to be put in when trying to get a better performance for the remaining cases.

This paper was written by the author of this thesis under supervision of the co-authors.

1.3.4 Embedding a Web-Based Workflow Management System in a Functional Language

Extending the iTask system with client-side processing is an important object of study for this thesis. This paper gives an overview of the experiences in developing the iTask system and shows which techniques are used in implementing it.

Workflow Management Systems (WFMS) are computer applications that coordinate, generate, and monitor tasks performed by human workers and computers. Workflow *specification* plays a dominant role in WFMSs: the work that needs to be done to achieve a certain goal is specified as a structured and ordered collection of tasks that are assigned to available resources at run-time. In many WFMSs, the workflow specification only provides an execution environment framework for the workflow that has to be complemented with custom code in a different programming formalism. In other WFMSs one has to provide much detail in the workflow specification itself. In both approaches substantial coding is required to complete the workflow application. In general, this results in complex distributed and heterogeneous applications that are hard to maintain.

The iTask system is developed to overcome these and other issues. The iTask system is a *domain specific language* that is *embedded* in the general purpose *programming language* Clean as a workflow specification language. This approach has a number of important advantages. In particular, these are the availability of the strong type system, of higher-order functions, of lazy and strict evaluation, and of the module system. All computational and algorithmic concerns can be dealt with in the Clean language. The domain specific language inherits these features from the embedding language. Adding these features to a stand-alone domain specific language would be a huge effort.

The iTask system is built on a single, powerful, concept: the *task*. The system uses combinators to combine tasks into new tasks. With combinators tasks can be executed sequentially or in parallel using or-, and- and ad-hoc parallelism. In a workflow specification a task is seen as a black-box unit: something that has to be done. It does not matter how it is executed, but if it is finished you can use its result (in other tasks). iTask has a number of predefined primitive tasks, but, if necessary, it is also possible to add new primitive tasks to the system. Examples of such tasks are: exchanging information with web-services or relational data bases; displaying information like charts or maps, etc. In this way the iTask system can also be considered a web-coordination language and therefore as a significant extension of the host language Clean.

iTask can also be considered as a declarative language. The user only has to specify the data types involved and the control of the information flow. All boilerplate code generation is taken care of as much as possible. As an example, interactive web-forms for user data acquisition are generated automatically from algebraic data types, and also the handling of data entered by the user is done automatically. This is realized by the use of generic functions [Hin00]. Generic functions are also used for many other issues like the storage of information by the server application. iTask can therefore also be seen as an advanced example of the use generic programming

techniques. The iTask concept was originally developed by Plasmeijer et al [PAK07].

This paper was a combined collaborative effort of all authors.

1.3.5 Declarative Ajax and Client-Side Evaluation of Workflows using iTasks

The original iTask system was a thin client application (see Fig. 1.1). This means that all processing is done at the server side of the application and that every user action on the client leads to a complete client-server round trip and to the generation of a completely new web-page. This results in less responsive applications. The Ajax paradigm offers two ways to tackle this problem (see Fig. 1.2): client-side processing and partial updates of web-pages. This paper describes how this is realized in the context of the iTask system. An important condition for this implementation is that the highly declarative nature of the iTask system and the generation of the complete application from a single source in Clean should be maintained. As a consequence of this, all client-side processing is to be realized by either generic or generated JavaScript or done using a Clean platform at the client side. We have chosen a combination of a client-side Clean platform with generic JavaScript code to glue everything together.

To implement a client-side Clean platform a Java Applet version of the SapI interpreter was created together with a Core Clean to SapI compiler (integrated in the back-end of the Clean compiler). Core Clean is the intermediate language used by the Clean compiler. Java Applet execution is available for all popular web-browsers thus offering an easy accessible platform for client-side processing.

The implementation of the iTask system is based on the repetitive rewriting of a task tree which represents the current state of the iTask application. The nodes in a task tree correspond to iTask combinators, whereas the leaves correspond to primitive tasks. Nodes in the task tree are overwritten by their result as soon as the corresponding task is finished. In the original implementation of iTask the entire task tree for an application is reconstructed after each user event. For this, information about the current state of the task tree is maintained in a combination of server and client-side (web-page) storage. An iTask application finishes after processing each user event and is re-executed for each new event. For Ajax and client-side processing we implemented (client-side) local task-tree rewriting. Instead of rewriting the entire task tree, only that part of the task tree corresponding to the current (sub)task is rewritten. With the result a (partial) update of the web page is created. This saves the time needed to reconstruct the entire task tree and in case of client-side processing also a client-server round trip.

To realize local task-tree rewriting we need the function that is capable of handling events for the specific subtree. This can be realized in Clean with the use of Dynamics. By using Dynamics, instances of any type, including function types, can be stored and even be exchanged between independently programmed Clean applications [Pil99, Wee07], while keeping the advantages provided by a strongly typed programming environment. Here we use Dynamics to serialize functions that can handle events for subtrees and to store them in a table. Events are encoded in such

a way that they can easily be used to retrieve the correct task handling function from this table. For client-side local task-tree rewriting we extended Clean Dynamics with a Sapl variant of it: Clean-Sapl Dynamics. Using this variant one can serialize an arbitrary Clean function (actually a closure or partial function application) in a Clean application to a string value, transport this string to the corresponding Sapl version of the application, and use the function there to handle task events.

By default, tasks that are assigned to a specific user are always implemented using local (server-side) task-tree rewriting. Client-side rewriting can be enforced by attaching the `OnClient` annotation to a task. If for some reason the task cannot be executed at the client side, for example because server-side stored information is needed, the system automatically falls back to server-side task rewriting.

The writing of this paper was a combined collaborative effort of all authors. For the technical part the author of this thesis realized: the Java Applet version of the Sapl interpreter; the Core Clean to Sapl compiler; the Sapl generation part of the Clean-Sapl Dynamics.

1.3.6 iEditors: Extending iTask with Interactive Plug-ins

Plug-ins are used to extend internet applications with functionality that is not offered by standard Html elements. Examples of plug-ins are media players that are used to play video, sound and animations, rich text editors for the creation of Html, etc. Another example are Java Applets that are used to embed Java applications in web-pages. In this paper we show how we can extend iTask applications with plug-ins. Although we focus on Java Applet plug-ins, many of the ideas used for incorporating them in iTask applications also apply to plug-ins written in other languages.

The natural way to look at a plug-in from an iTask point of view is that of a primitive task. A plug-in is used to perform some work: to play a video; to edit a text, etc. Of course, it should be possible to supply the plug-in with data generated by other tasks and to use data generated by the plug-in in subsequent tasks.

What are the technical issues to be dealt with when incorporating a plug-in in an iTask application? First of all a plug-in should be loaded into the web browser. The more challenging issue is the exchange of information with a plug-in. If we use a plug-in to edit a text, we should supply the plug-in with the text to be edited (maybe generated by a previous task) and, after editing is finished, we have to be able to get access to the edited text (and to use it in subsequent tasks).

iTask uses generic functions to generate forms from data types and to process user data entered in these forms. By specializing these functions for certain types one can obtain dedicated behavior, e.g. a dedicated form. This is exactly what we need for including plug-ins in iTask. By using a plug-in wrapper type and specializing the generic form generation function for this type, the correct Html or JavaScript representation for the plug-in can be generated and the plug-in can be supplied with correctly formed input data. By specializing the generic function that handles form data for this type, the data edited in the plug-in can be converted back to the corresponding Clean type.

Sometimes, a plug-in needs specific further processing for generating the correct

result. An example of such a plug-in is a graphical (diagram) editor. Depending on the kind of editor made, the editor should react in a specific way on mouse (up, down and drag) events. The standard way to realize this, is to make a dedicated editor plug-in for each kind of data one needs to edit. In this paper we demonstrate that it is also possible to make a generic kind of editor and to do all specific processing in Clean. This is realized by attaching call-back functions in Clean for handling specific events to the plug-in. The call-back function is used to process events and generate new data for the plug-in. It is even possible to handle events at the client side by using the SapI interpreter and the Clean to SapI compiler. Again Clean-SapI Dynamics [Pil99, Wee07] is used to serialize the call-back function and to transport it from server to client. In this way it is even possible to create sophisticated graphical editors, where all specific processing is done in Clean at the client side.

This paper was written by the author of this thesis under supervision of the co-authors.

1.3.7 Web Based Dynamic Workflow Systems for C2 of Military Operations

Military and crisis-management operations involve cooperation and collaboration between a large number of diverse organizations. Activities in these operations are highly dynamic and situation dependent. To cooperate and collaborate, activities performed by diverse organizations must be synchronized (or at least de-conflicted). A dynamic workflow tool-kit can therefore be helpful in the development of applications for the military and crisis-management domain.

This paper presents an initial discussion on the suitability of dynamic workflow specifications, and its implementation in the iTask system, for military and crisis-management operations. The discussion is based on five key design requirements for response technology [Jul07]: suitability for just in time learning; response drivenness; support for co-operation between parties involved; adaptability and flexibility; and robustness against failure.

In the paper it is shown that the iTask system already meets important aspects of these requirements, and can be trivially extended to meet even more. Especially with respect to adaptability and flexibility the iTask shows great potential. For example, iTask can deal with dynamic behavior in several ways. First, iTask workflows are data driven, so new tasks can dynamically depend on the results of previous tasks. Second, iTask supports an exception mechanism that makes it possible to stop running workflows in case an unexpected situation occurs. Third, iTask can replace a task by another task in a running workflow and in this way can cope with chancing circumstances.

However, the evaluation also gives insight in the research challenges that need to be addressed to fully optimize the iTask system for supporting military and crisis response operations. The areas we identified are: better support for collaboration between different parties involved; obtaining information about the current state of a workflow to be able to adapt the workflow; the creation of domain specific frameworks to enable the rapid development of workflow applications. Although

some of the strengths, weaknesses and challenges discussed apply only to the iTask system, most apply to workflow management systems in general.

This paper was written by the first two authors under supervision of the third and fourth author.

1.3.8 Conclusions and Discussion

In this chapter we reflect on our work, look at related work, discuss a number of issues that came up after the papers presented in the previous chapters were finished and sketch future research for (applications of) the iTask system.

We start with a more extensive discussion on the efficiency of the SapI interpreter and give some suggestions how this efficiency can be improved even further. We argue that standard compiler optimizations techniques based on strictness analysis and tail recursion do not give much benefit for interpreters. Instead, using the SapI compiler to compile frequently used functions (e.g. standard libraries) and add them as C++ code to the interpreter, can result in significant speed-ups for many programs.

The second issue for discussion is the use of Clean as a platform to embed other domain specific languages in. Modern functional programming languages like Haskell and Clean are more than just programming languages. In fact they are tool building languages that allow for the quick development of new programming formalisms.

The third issue for discussion is the use of Java as implementation platform for the client-side version of the SapI interpreter. We discuss whether JavaScript is a useful alternative for Java.

The fourth discussion is about iTask as a programming platform. We argue that iTask is more than just a dynamic workflow language, but can also be used as a web integration tool.

The last discussion is about alternative application areas for iTask.

Chapter 2

Comprehensive Encoding of Data Types and Algorithms in the λ-Calculus

[1] ***Abstract*** The λ-calculus is a well known basic universal programming language, but is not considered as a realistic option for expressing algorithms in a comprehensive way. In this paper we show that this poor reputation is mainly caused by the choice of the Church encoding for the representation of Algebraic Data Types. We show that, using a different encoding attributed to Scott, and with a little aid of a clever lay-out scheme, functional programs, like those written in languages like Clean or Haskell, can be expressed using comprehensive and concise λ-expressions resembling their Haskell and Clean counterparts. For this purpose, we also use an alternative way to express recursion without the use of a fixed-point combinator. The resulting formalism not only allows for comprehensive and readable code, but also allows for an efficient implementation.

2.1 Introduction

Although the λ-calculus is considered to be the mother of all (functional) programming languages, programming in it is not considered to be very practical. Every course or textbook on λ-calculus (e.g. [Bar84]) spends some time on showing how the well-known programming constructs can be represented in the λ-calculus. It commonly starts by explaining how to represent data types like natural numbers in the λ-calculus and how to define operations on them. In almost all cases the Church numerals are chosen as leading example. The definition of Church numerals and operations on them shows that it is possible to use the λ-calculus for all kinds of computations and that it is indeed a universal programming language. The Church encoding can be generalized for the encoding of general Algebraic Data Types (see [Bar97]). This encoding allows for a straightforward implementation of iterative (primitive recursive) or fold-like functions on data structures, but needs complex and inefficient constructions for expressing general recursion. In this way one ends

[1] Submitted as [JPK10]

up with an encoding that works in theory but is also quite unreadable (and inefficient).

It is less commonly known that there exist alternative encodings of numbers in the λ-calculus. In [JKP06] (chapter 3) we already introduced an alternative encoding for Algebraic Data Types and showed that this encoding allows for an efficient implementation of interpreters for functional languages with data types based on this encoding. While in our previous work the focus was on obtaining an efficient interpreter for an intermediate functional language, here we have an entirely different goal. We look at the λ-calculus from a programmers perspective and want to show that the use of this encoding also makes it possible to obtain comprehensible λ-expressions for the realization of data structures and algorithms.

Another issue to be dealt with when using the λ-calculus as a programming language is the representation of recursive functions. Because λ-expressions are nameless, they cannot refer to themselves, and a special construction is needed to express recursion. The standard way to do this is the use of a fixed point combinator. Here we show that we can express recursion without the use of fixed point combinators, with as only price a small change in the way recursive functions are called. A further gain of this representation of recursion is that it results in a more efficient implementation using fewer reduction steps than when using a fixed point combinator.

This paper is organized as follows: We start with describing the Scott encoding in Section 2.2. In Section 2.3 we sketch how recursion can be expressed without using a fixed-point combinator. In Section 2.4 we show how we can use the techniques from the previous sections to express complete programs as a single λ-expression. We make a comparison of the Scott and Church encodings in Section 2.5 and end with some conclusions in Section 2.6.

2.2 Alternative Encoding of Algebraic Data Types

The encoding we use is relatively unknown, and independently (re)discovered by several authors (e.g. [SM89, Mog94, Stu08] and the first author), but originally attributed to Scott in an unpublished lecture which is cited in Curry, Hindley and Seldin ([CHS72], page 504) as: *Dana Scott, A system of functional abstraction. Lectures delivered at University of California, Berkeley, Cal., 1962/63. Photocopy of a preliminary version, issued by Stanford University, September 1963, furnished by author in 1968.*[2] We will therefore call it the *Scott* encoding. The encoding results in a representation that is very close to algebraic data types as they are used in most functional programming languages. We illustrate this with some examples of well-known data types.

[2]We would like to thank Matthew Naylor for pointing us to this reference.

2.2.1 The Nature of Algebraic Data Types

Consider Algebraic Data Type (ADT) definitions in languages like Clean or Haskell such as tuples, booleans, temperature, maybe, natural (Peano) numbers, and lists:

```
data Boolean     = True | False
data Tuple a b   = Tuple a b
data Temperature = Fahrenheit Int | Celsius Int
data Maybe a     = Nothing | Just a
data Nat         = Zero | Suc Nat
data List t      = Nil  | Cons t (list t)
```

A type consists of one or more alternatives. Each alternative consist of a name, possibly followed by a number of arguments. Algebraic Data Types are used for several purposes:

- to make enumerations, like in `Boolean`;
- to package data, like in `Tuple`;
- to unite things of different kind in one type, like in `MayBe` and `Temperature`;
- to make recursive structures like in `Nat` and `List` (in fact to construct new types with an infinite number of elements).

The power of the ADT construction in modern functional programming languages is that one formalism can be used for all these purposes. Imperative formalisms like C and Java need several constructs (like enumeration types, records, pointers and inheritance) for achieving this. Algebraic Data Types also have a meaning in untyped and dynamically typed formalisms like Lisp. But in that case the packaging concept is the most important one. The packaging construct is needed for the assembly of composed results for functions and for the construction of arbitrary size data containers. Lisp uses the list as a kind of generic packaging construct.

If we analyse the construction of ADT's more carefully, we see that constructor names are used for two purposes. First, they are used to distinguish the different cases in a single type definition (like `True` and `False` in `Boolean` and `Fahrenheit` and `Celsius` in `Temperature`). Second, we need them for recognizing them as being part of a type and making type inferencing possible. Therefore, all constructor names must be different in a single functional program (module). For distinguishing the different cases in a function definition, pattern matching on constructor names is used.

In the next three subsections we show how ADT's can be expressed as λ-expressions in a natural way, staying close to their original definitions.

2.2.2 Named λ-expressions

First, some remarks about the notation of λ-expressions. We will always give a λ-expression representing an ADT or a function a name:

True $\equiv \lambda$`a b . a`

In this way it is possible to refer to this λ-expression. If in a λ-expression a name is in italics, then it refers to another λ-expression having this name. This is done for readability and for saving space. For example:

True (λf g . f g) (λf g . g f)

Should be read as:

(λa b . a) (λf g . f g) (λf g . g f)

By introducing an additional λ-abstraction and using the fact that (λtrue . true y z) x reduces to x y z, we can also write:

(λtrue . true (λf g . f g) (λf g . g f)) (λa b . a)

The last example shows a well known alternative way of introducing explicit names in λ-expressions (see also Section 2.4).

Named λ-expressions are only introduced for notational convenience. These definitions behave like macro definitions. The names are replaced by the corresponding body before any reduction is done. This implies that these definitions cannot be recursive.

2.2.3 Expressing Enumerations Types in the λ-calculus

The simplest example of such a type is Boolean. We already noted that we use pattern matching for recognizing the different cases (constructors). So we are actually looking for an alternative for pattern matching using λ-expressions. The simplest example of using a pattern match for booleans is the if-then-else construction:

```
ifte True  a b = a
ifte False a b = b
```

But the same effect can easily be achieved by making True and False functions, selecting the left or right argument respectively and by making ifte the identity function. Therefore, the λ-calculus solution for this is straightforward:

True ≡ λa b . a
False ≡ λa b . b
ifte ≡ λt . t

This is also the standard (Church) encoding used for booleans in λ-calculus courses and text books. So far we learned nothing new yet!

2.2.4 Expressing a Simple Container Type in the λ-calculus

Tuple is the simplest example of a pure container type. If we group data into a container type, we also need constructions to get data out of the container (so-called projection functions). For Tuple this can be realized by pattern matching or by using the selection functions fst and snd. These functions can be defined in Haskell using pattern matching:

```
fst (Tuple a b) = a
snd (Tuple a b) = b
```

Containers can be expressed in the λ-calculus by using closures (partial applications). For `Tuple` the standard way to do this is:

Tuple $\equiv \lambda$`a b f . f a b`

A tuple is a function that takes 3 arguments. If we supply only two, we have a closure. This closure can take a third argument, which should be a 2 argument function. This function is then applied to the first two arguments. The third argument is therefore called a continuation (the function with which the computation continues). It is now easy to find out what the definitions of `fst` and `snd` should be:

fst $\equiv \lambda$`t . t (`λ`a b . a)`
snd $\equiv \lambda$`t . t (`λ`a b . b)`

If applied to a tuple, they apply the tuple to a two argument function, that selects either the first (`fst`) or second (`snd`) argument.

Again, this definition of tuples is the one that can be found in λ-calculus text books and courses. So again, we learned nothing new.

2.2.5 Expressing General Multi Case Types in the λ-calculus

It is now a simple step to come up with a solution for arbitrary ADT's. Just combine the two solutions from above. Let us look at the definition of the function `warm` that takes a `Temperature` as an argument:

`warm :: Temperature` $\rightarrow$ `Boolean`
`warm (Fahrenheit f)` $=$ `f` > 90
`warm (Celsius c)` $=$ `c` > 30

We have to find encodings for (`Fahrenheit f`) and (`Celsius c`). The first solution tells that we should make a λ-expression with 2 arguments that returns the first argument for `Fahrenheit` and the second argument for `Celsius`. The second solution tells that we should feed the argument of `Fahrenheit` or `Celsius` to a continuation function. Combining these two solutions we learn that `Fahrenheit` and `Celsius` should both have 3 arguments. The first one to be used for the closure and the second and third as continuation arguments. `Fahrenheit` should choose the first continuation argument and apply it to its first argument and `Celsius` should do the same with the second continuation argument. So their definitions now become:

`Fahrenheit` $\equiv \lambda$`i` f_c c_c `. ` f_c `i`
`Celsius` $\equiv \lambda$`i` f_c c_c `. ` c_c `i`

The definition of `warm` now becomes:

warm $\equiv \lambda$`t . t (`λ`f . f` > 90`) (`λ`c . c` > 30`)`

If we apply this strategy to the types `Nat` and `List` we obtain the following definitions for the constructors:

Zero $\equiv \lambda z_c\ s_c$ `. ` z_c
Suc $\equiv \lambda$`n` $z_c\ s_c$ `. ` s_c `n`

Nil $\equiv \lambda n_c\ c_c$ `. ` n_c

$Cons \equiv \lambda \mathtt{x\ xs\ n}_c\ \mathtt{c}_c\ .\ \mathtt{c}_c\ \mathtt{x\ xs}$

Note that in these definitions the fact that these data types are recursive is of no influence. Functions like predecessor, head and tail can now easily be defined:

$pred \equiv \lambda \mathtt{n}\ .\ \mathtt{n}\ undef\ (\lambda \mathtt{m}\ .\ \mathtt{m})$

$head \equiv \lambda \mathtt{xs}\ .\ \mathtt{xs}\ undef\ (\lambda \mathtt{x\ xs}\ .\ \mathtt{x})$
$tail \equiv \lambda \mathtt{xs}\ .\ \mathtt{xs}\ undef\ (\lambda \mathtt{x\ xs}\ .\ \mathtt{xs})$

pred and ***tail*** are here constant time functions, while in the Church encoding their definitions are linear in the size of `n` or `xs` (see Sec. 2.5). In partial functions like ***head***, ***pred*** and ***tail*** we use ***undef*** to indicate the part of the function that is not defined.

2.2.6 The General Case

In general the mapping of an ADT to λ-expressions is defined as follows. Given the following ADT definition in Haskell or Clean:

$\mathtt{data\ type\_name\ t}_1\ \ldots\ \mathtt{t}_k = \mathtt{C}_1\ \mathtt{t}_{1,1}\ \ldots\ \mathtt{t}_{1,n_1}\ \mid\ \ldots\ \mid\ \mathtt{C}_m\ \mathtt{t}_{m,1}\ \ldots\ \mathtt{t}_{m,n_m}$

Then this type definition with m constructors can be mapped to m λ-expressions:

$\mathtt{C}_1 \equiv \lambda \mathtt{v}_{1,1}\ \ldots\ \mathtt{v}_{1,n_1}\ \mathtt{f}_1\ \ldots\ \mathtt{f}_m\ .\ \mathtt{f}_1\ \mathtt{v}_{1,1}\ \ldots\ \mathtt{v}_{1,n_1}$
$\ldots$
$\mathtt{C}_m \equiv \lambda \mathtt{v}_{m,1}\ \ldots\ \mathtt{v}_{m,n_m}\ \mathtt{f}_1\ \ldots\ \mathtt{f}_m\ .\ \mathtt{f}_m\ \mathtt{v}_{m,1}\ \ldots\ \mathtt{v}_{m,n_m}$

Consider the (multi-case) pattern-based function `f` in Haskell or Clean defined on this type:

$\mathtt{f}\ (\mathtt{C}_1\ \mathtt{v}_{1,1}\ \ldots\ \mathtt{v}_{1,n_1}) = \mathtt{body}_1$
$\ldots$
$\mathtt{f}\ (\mathtt{C}_m\ \mathtt{v}_{m,1}\ \ldots\ \mathtt{v}_{m,n_m}) = \mathtt{body}_m$

This function is converted to the following λ-expression using the Scott encoding of data types:

$$f \equiv \lambda \mathtt{x}\ .\ \mathtt{x}\ (\lambda \mathtt{v}_{1,1}\ \ldots\ \mathtt{v}_{1,n_1}\ .\ \mathtt{body}_1)\ \ldots\ (\lambda \mathtt{v}_{m,1}\ \ldots\ \mathtt{v}_{m,n_m}\ .\ \mathtt{body}_m)$$

This completes the description of the Scott encoding of data types. In section 2.5 we compare the Scott representation with the widely used Church encoding of data types.

2.3 Defining Recursive functions

Now we know how to represent ADT's we can concentrate on functions. We already gave some examples of them above (`ifte`, `fst`, `snd`, `head`, `tail`, `pred`, `warm`). The more interesting examples are the recursive functions. The standard technique for defining a recursive function in the λ-calculus is with the use of a fixed point operator. Let

us look for example at the addition operator for Peano numbers. In Haskell or Clean we express this by:

```
add Zero     m = m
add (Suc n) m = Suc (add n m)
```

Using the Scott encoding and recursion in the definition, this becomes:

$add_0 \equiv \lambda$n m . n m (λn . *Suc* (*add*$_0$ n m))

This definition is illegal because it uses a reference to the macro add_0 itself. With the use of the Y fixed point combinator to eliminate recursion this becomes:

$add_Y \equiv Y$ (λadd n m . n m (λn . *Suc* (add n m)))
$Y \equiv \lambda$h . (λx . h (x x)) (λx . h (x x))

There is, however, another way to represent recursion. Instead of using a fixed point operator we can also give the recursive function itself as an argument (like this is done in the argument of Y in add_Y):

add $\equiv \lambda$add n m . n m (λn . *Suc* (add add n m))

The price to pay is that each call of add should have add as an argument, as can been seen in the definition of *add*. The gain is that we do not need the fixed point operator any-more and that we can recognize recursive calls on the spot. This definition is also more efficient than the one with the fixed-point combinator, because it uses fewer reduction steps for evaluation when using normal order reduction. The following example shows how *add* can be used to add one to one:

(λadd . add add (*Suc Zero*) (*Suc Zero*)) *add*

2.3.1 Mutually Recursive functions

In case of mutually recursive functions, we have to add all mutually recursive functions as arguments for each function in the mutual recursion. An example to clarify this (we start with the Haskell definitions):

```
isOdd Zero     = False
isOdd (Suc n)  = isEven n
isEven Zero    = True
isEven (Suc n) = isOdd n
```

This can be represented by λ-expressions as:

isOdd $\equiv \lambda$isOdd isEven n . n *False* (λn . isEven isOdd isEven n)
isEven $\equiv \lambda$isOdd isEven n . n *True* (λn . isOdd isOdd isEven n)

All mutually recursive functions are now an argument of all functions in the definition as well as in each applied occurrence.

2.4 Converting Clean and Haskell Programs to λ-calculus

We now have all ingredients ready for converting complete programs. The last step to be made is combining everything into a single λ-expression. For example, if we take the `add 1 1` example from above, and substitute all macros, we obtain:

```
(λadd . add add ((λn f g.g n) (λf g.f)) ((λn f g.g n) (λf g.f)))
        (λadd n m . n m (λn . (λn f g.g n) (add add n m)))
```

Using ordinary β-reductions this reduces to a term equivalent to *Suc (Suc Zero)* that represents the desired value 2. As said before, we can introduce explicit names for `zero` and `suc` by abstracting out their definitions and obtain a more comprehensive definition:

```
(λzero suc .
    (λadd  .
            add add (suc zero) (suc zero))
    (λadd n m . n m (λn . suc (add add n m))))
(λf g.f) (λn f g.g n)
```

Here we applied a kind of inverted λ-lifting (see Sec. 2.4.2). We have used some smart indentation to make the expression better readable. The main expression is indented most. Definitions are introduced by variable names before they are used. Their implementations are indented as much as the line where their names were introduced. Note the nesting in this definition: the definition of `add` is inside the scope of the variables `suc` and `zero`, because its definition depends on the definition of them. In this way the macro reference *Suc* in the definition of `add` can be replaced by a variable `suc`.

As another example, the right hand side of the Haskell function:

```
main = isOdd (Suc (Suc (Suc Zero)))
```

can be written as:

(λisOdd isEven . isOdd isOdd isEven (*Suc* (*Suc* (*Suc Zero*)))) *isOdd isEven*

and after substituting all macro definitions and applying inverted λ-lifting:

```
(λtrue false zero suc .
            (λisOdd isEven .
                    isOdd isOdd isEven (suc (suc (suc zero))))
            (λisOdd isEven n . n false (λn . isEven isOdd isEven n))
            (λisOdd isEven n . n true  (λn . isOdd isOdd isEven n)))
(λa b.a) (λa b.b) (λf g.f) (λn f g.g n)
```

The conversion yields small λ-terms in which the original functional version of the definition is easily recognizable.

2.4.1 Some Remarks on the Evaluation of Expressions

We use normal order reduction for the λ-expressions to achieve lazy evaluation similar to lazy functional languages like Haskell and Clean. In order to obtain recognizable results we treat λ-expressions like:

$\lambda \mathtt{x}_1\ \mathtt{x}_2\ \ldots\ \mathtt{x}_n\ .\ \mathtt{e}$

not as an abbreviation of:

$\lambda \mathtt{x}_1\ .\ (\lambda \mathtt{x}_2\ .\ (\ldots\ \lambda \mathtt{x}_n\ .\ \mathtt{e}\ \ldots))$

as usually in the λ-calculus. In contrast we have a special reduction rule for each `n`:

$(\lambda \mathtt{x}_1\ \ldots\ \mathtt{x}_n\ .\ \mathtt{e})\ \mathtt{a}_1\ \ldots\ \mathtt{a}_n \equiv (\ldots(\mathtt{e}\ [\mathtt{x}_1 = \mathtt{a}_1])\ldots)\ [\mathtt{x}_n = \mathtt{a}_n]$

That is, only if the λ-expression has all its arguments, it is reduced as an ordinary λ-expression. Without the proper number of arguments no reduction steps are applied (exactly the reduction behavior of Clean and GHC).

As a consequence, (λ`n f g . g n`) (λ`f g . f`), representing *Suc Zero*, is not considered to be a redex and will therefore not be reduced to λ`f g . g` (λ`f g . f`).

2.4.2 Formalizing Inverted λ-lifting

Above we mentioned the operation of inverted λ-lifting. Here we make more precise what we mean by this. The conversion of a functional program from Clean or Haskell into a λ-expression proceeds in a number of steps:

1. Remove all syntactic sugar (list notation, zf-expressions, `where` and `let` expressions, etc.).
2. Eliminate all algebraic data type definitions by converting them to functions using the Scott encoding.
3. Convert pattern-based function definitions to normal functions using the Scott encoding of algebraic data types (see Sect. 2.2.6).
4. Remove (mutually) recursion by the introduction of extra variables (as explained in Sec. 2.3).
5. Make a dependency sort of all functions, resulting in an ordered collection of sets (strongly connected components). So the first set contains the functions that do not depend on other functions (e.g. the Scott encoded ADT's). The second set contains the functions that only depend on the functions in the first set, etc. Hereby, a group of mutually recursive functions is treated as a single function and thus all functions in it must belong to the same dependency set. Note that we can do this because all possible cycles are already removed in the previous step.
6. Construct the resulting λ-expression by nesting the definitions from the different dependency sets. The outermost expression consists of an application of a λ-expression with as variables the names of the functions from the first dependency set and as arguments the λ-definitions of these functions. The body of this expression is obtained by repeating this procedure for the remainder dependency sets. The innermost expression is the main expression.

The result of this process is:

```
(λfunction_names_first_set .
  (λfunction_names_second_set .
    ...
      (λfunction_names_last_set .
        main_expression)
      function_definitions_last_set)
    ...
  function_definitions_second_set)
function_definitions_first_set
```

2.4.3 A More Complex Example

As a last, more interesting example, consider the following Haskell version of the Eratosthenes prime sieve program:

```
data Nat                  = Zero | Suc Nat
data Inflist t            = Cons t (Inflist t)
nats n                    = Cons n (nats (Suc n))
sieve (Cons Zero xs)      = sieve xs
sieve (Cons (Suc k) xs)   = Cons (Suc k) (sieve (rem k k xs))
rem p Zero     (Cons x xs) = Cons Zero (rem p p xs))
rem p (Suc k)  (Cons x xs) = Cons x    (rem p k xs)

main = sieve (nats (Suc (Suc Zero)))
```

Here we use infinite lists for the storage of numbers and the resulting primes. `sieve` filters out the `zero`'s in a list and calls `rem` to set multiples of prime numbers to `zero`. Applying the first four steps of the conversion procedure results in:

```
Zero  ≡ λf g     . f
Suc   ≡ λn f g   . g n
Cons  ≡ λx xs g . g x xs
nats  ≡ λnats n .  Cons n (nats nats (Suc n))
sieve ≡ λsieve ls . ls (λx xs . x (sieve sieve xs)
                                  (λk .  Cons x (sieve sieve (rem rem k k xs))))
rem   ≡ λrem p k ls . ls (λx xs . k (Cons Zero (rem rem p p xs))
                                   (λk .  Cons x (rem rem p k xs)))
main ≡ sieve sieve (nats nats (Suc (Suc Zero)))
```

The dependency sort results in:

```
[{zero,suc,cons},{rem,nats},{sieve},{main}]
```

Putting everything together in a single λ-expression yields:

```
(λzero suc cons .
 (λrem nats      .
  (λsieve        .
    sieve sieve (nats nats (suc (suc zero))))
  sieve)
 rem  nats)
Zero Suc Cons
```

And after substituting the λ-definitions for all macros:

```
(λzero suc cons .
 (λrem nats     .
  (λsieve       .
    sieve sieve (nats nats (suc (suc zero))))
  (λsieve ls  . ls (λx xs . x (sieve sieve xs)
                              (λk . cons x (sieve sieve (rem rem k k xs))))))
 (λrem p k ls . ls (λx xs . k (cons zero (rem rem p p xs))
                              (λk . cons x (rem rem p k xs))))
 (λnats n . cons n (nats nats (suc n))))
(λf g . f) (λn f g . g n) (λx xs g . g x xs)
```

Which is probably the most compact, completely self-contained, definition of a prime number generator. Even shorter (143 characters) using one letter identifiers:

$$(\lambda zsc.(\lambda rf.(\lambda e.ee(ff(s(sz))))(\lambda el.l\lambda ht.h(eet)\lambda k.ch(ee(rrkkt))))$$
$$(\lambda rpkl.l\lambda ht.k(cz(rrppt))\lambda k.ch(rrpkt))(\lambda fn.cn(ff(sn))))(\lambda fg.f)(\lambda nfg.gn)(\lambda htg.ght)$$

2.5 Comparing the Church and Scott encoding

We already indicated that the Church and Scott encoding overlap for simple enumerations and simple (non-recursive) packaging types. They only differ for recursive types. Let us have a look at the Church definition of natural numbers:

$Zero_c \equiv \lambda$`f x . x`
$Suc_c \equiv \lambda$`n f x . f (n f x)`

As a reminder, above we had for the Scott encoding:

$Zero_s \equiv \lambda$`f g . f`
$Suc_s \equiv \lambda$`n f g . g n`

The functions $Zero_c$ and $Zero_s$ are both selection functions, but the definition of Suc_c is completely different from Suc_s. Instead of feeding only n to the continuation function `f` the result of of `n f x` is fed to the continuation function `f`. This is exactly the same thing as what happens in the `fold` function. Using the Scott encoding for natural numbers `fold` can be defined as (the recursion can be removed with the technique used earlier):

$foldNat \equiv \lambda$`f z n . n z (λn . f (`$foldNat$ `f z n))`

In [Hin05] Hinze states that Church numerals are actually folds in disguise. As a consequence only primitive recursive functions on numbers can be easily expressed using the Church encoding. An example of such a function is addition:

$add_c \equiv \lambda$`n m . n` Suc_c `m`

Which is comparable to the following Scott version using `foldNat`:

$add_s \equiv \lambda$`n m .` $foldNat$ Suc_s `n m`

For functions that need general recursion (or functions for which the result for `suc n` cannot be expressed using the result for `n`) we run into troubles. Church himself was not able to solve this problem but Kleene found a way out (as described in [Bar97]). A nice example of his solution is the predecessor function, which can be easily expressed using the Scott encoding, as we saw earlier:

$pred_s \equiv \lambda \texttt{n} \;.\; \texttt{n}\ undef\ (\lambda \texttt{m} \;.\; \texttt{m})$

To define it using the Church encoding Kleene used a construction with pairs.

$pred_c \equiv \lambda \texttt{n} \;.\; snd(\texttt{n}\ (\lambda \texttt{p} \;.\; pair\ (Suc_c\ (fst\ \texttt{p}))\ (fst\ \texttt{p}))\ (pair\ Zero_c\ Zero_c))$

Each pair combines the result of the recursive call with the previous element. A disadvantage of this solution, besides that it is hard to comprehend, is that $pred_c$ `n` has complexity *O(n)* while that of $pred_s$ `n` is *O(1)*. From a programmers point of view this is a serious drawback.

It is straightforward to convert Church and Scott encoded numbers into each other:

$toChurch \equiv \lambda \texttt{n f x} \;.\; foldNat\ \texttt{f x n}$
$toScott \equiv \lambda \texttt{n} \;.\; \texttt{n}\ Suc_s\ Zero_s$

This again, shows that the difference between the Church and Scott encoding is a fold!

2.5.1 Comparing the Scott and Church encoding for lists

The Church encoding for lists together with the functions `sum` and `tail` are given by:

$Nil_c \equiv \lambda \texttt{f x} \;.\; \texttt{x}$
$Cons_c \equiv \lambda \texttt{h t f x} \;.\; \texttt{f h (t f x)}$

$sum_c \equiv \lambda \texttt{xs} \;.\; \texttt{xs}\ add_c\ Zero_c$
$tail_c \equiv \lambda \texttt{xs} \;.\; snd\ (\texttt{xs}\ (\lambda \texttt{x rs} \;.\; pair\ (Cons_c\ \texttt{x}\ (fst\ \texttt{rs}))\ (fst\ \texttt{rs}))\ (pair\ Nil_c\ Nil_c))$

Also here the definition of `Cons` behaves like a fold (a `foldr` actually). Again, we need the pair construction for the non-primitive recursive function `tail`. The Scott version of `foldr` for lists and its application in the `sum` function are:

$foldList \equiv \lambda \texttt{f d xs} \;.\; \texttt{xs d}\ (\lambda \texttt{h t} \;.\; \texttt{f h}\ (foldList\ \texttt{f d t}))$
$sum_s \equiv \lambda \texttt{xs} \;.\; foldList\ add_s\ Zero_s\ \texttt{xs}$

The conversions between the Church and Scott encoding for lists are given by:

$toChurchList \equiv \lambda \texttt{xs f d} \;.\; foldList\ \texttt{f d xs}$
$toScottList \equiv \lambda \texttt{xs} \;.\; \texttt{xs}\ Cons_s\ Nil_s$

Note that these definitions are completely equivalent to those for the conversion of numbers. They only use a different fold function in *toChurchList* and different constructors in *toScottList*.

2.5.2 Discussion

We already indicated that the Scott encoding just combines the techniques used for encoding booleans and tuples in the Church encoding as described in standard λ-calculus text books and courses. The Scott and Church encodings only differ for recursive types. A Church encoded type just defines how functions should be folded over an element of the type. A fold can be characterized as a function that replaces constructors by functions. The Scott encoding just packages information into a closure. Recursiveness of the type is not visible at this level. Of course, this is also the case for ADT's in functional languages, where recursiveness is only visible at the type level and not at the element level.

The representation achieved using the Scott encoding is equivalent to that of ADT definitions in modern functional programming languages and allows for an similar realization of functions defined on ADT's. Also the complexity (efficiency) of these functions is similar to their equivalents in functional programming languages. This in contrast to their counterparts using the Church encoding that sometimes have a much worse complexity. Therefore, from a programmers perspective the Scott encoding is better than the Church encoding.

A disadvantage of the Scott encoding of ADT's is that the resulting functions cannot be typed using standard HM type systems, while Church encoded ADT's can be neatly typed. The encoding of recursive functions in combination with the absence of ordinary combinators is too complicated for the standard HM type systems.

An interesting question now is: Why did it took so long before the Scott encoding was discovered and why is this encoding still relatively unknown? The encoding is simpler than the Church encoding and allows for a straightforward implementation of functions acting on data types. Of course, the way ADT's are represented in modern functional programming languages is rather new and dates from languages like ISWIM [Lan66], HOPE [BMS80] and SASL [Tur79] and this was long after the Church numerals were invented. Furthermore, ADT's are needed and defined by programmers, who needed an efficient way to define new types, which is rather irrelevant for mathematicians who are less concerned with an efficient implementation of algorithms.

In [JKP06] (chapter 3) we showed that this representation of functional programs can be used to construct very efficient, simple and small interpreters for lazy functional programming languages. These interpreters only have to implement β-reduction and no constructors nor pattern matching.

Altogether, we argue that the Scott encoding also should have its place in λ-calculus textbooks and courses.

2.6 Conclusions

In this paper we showed how the λ-calculus can be used to express algorithms and Algebraic Data Types in a way that is close to the way this is done in functional programming languages. To achieve this, we used a rather unfamiliar encoding of

ADT's attributed to Scott. We showed that this encoding can be considered as a logical combination of the way how enumerations (like booleans) and containers (like tuples) are normally encoded in the λ-calculus. The encoding differs from the Church encoding and the connecting element between them is the fold function.

For recursive functions we did not use the standard fixed-point combinators, but instead used a simple technique where an expression representing a recursive function is given (a reference to) itself as an argument. In this way the recursion is made more explicit and this also results in a more efficient implementation using fewer reduction steps.

We also sketched a systematic method for converting Haskell or Clean like programs to λ-expressions.

Altogether we have shown that it is possible to express a functional program in a concise way as a λ-expression that is clearer than the standard Church representation of the functional program.

Chapter 3

Efficient Interpretation by Transforming Data Types and Patterns to Functions

[1] ***Abstract*** This paper describes an efficient interpreter for lazy functional languages like Haskell and Clean. The interpreter is based on the elimination of algebraic data types and pattern-based function definitions by mapping them to functions using a new efficient variant of the Church encoding. The transformation is simple and yields concise code. We illustrate the concepts by showing how to map Haskell and Clean programs to the intermediate language Sapl (**S**imple **A**pplication **P**rogramming **L**anguage) consisting of pure functions only.

An interpreter is described for Sapl, based on straightforward graph-reduction techniques. This interpreter can be kept small and elegant because function application is the only operation in Sapl. The application of a few easy to realize optimisations turns this interpreter into an efficient one. The resulting performance turns out to be competitive in a comparison with other interpreters like Hugs, Helium, GHCi and Amanda for a large number of benchmarks.

3.1 Introduction

In this paper we present an implementation technique for lazy functional languages like Haskell [PJ03] and Clean [PE01] based on the representation of data types by functions. Although it is well known that it is possible to represent algebraic data types as functions by using the Church encoding or variants of it (Berarducci and Bohm ([BB93] and [BB85]) and Barendregt [Bar97]), these representations have never been used in implementations for efficiency reasons. Therefore, intermediate languages always contain special constructs for data types and pattern matching (see e.g. Peyton Jones [PJ87] and Kluge [Klu04]). In this paper we present a new variant of the Church encoding for algebraic data types. This variant uses named functions and explicit recursion instead of lambda expressions for the conversion. We

[1] Originally published as [JKP06]

show how to convert a pattern-based function definition to a single function without patterns using this encoding. The encoding results in a program in the intermediate language Sapl consisting of pure functions only. The encoding we use has important advantages over the Church encoding because it allows for destructor functions with complexity $O(1)$, instead of proportional to the size of the data structure (list, tree, etc.).

In the second half of this paper an interpreter is described that can handle the functions that are the result of this transformation. The interpreter is based on straightforward graph-reduction techniques. To optimise the performance of the interpreter two types of function annotations are introduced. The first annotation enables an optimal instantiation of function bodies that are the result of translating pattern-based function definitions, and the second annotation enables the inline execution of certain local function definitions. The annotations can easily be added during the translation of a Haskell or Clean program to Sapl. It is also possible to add them during a static analysis of the translated programs without knowledge of the original data types and pattern definitions.

Summarizing, the contributions of this paper are:

- We introduce a new encoding scheme that transforms algebraic data types to simple function definitions in the intermediate language Sapl. The encoding uses named functions and explicit recursion which simplify the encoding considerably in comparison with known encodings.

- We show how to transform a pattern-based function definition to a single function without patterns using this encoding.

- We describe how an efficient interpreter can be realized for lazy functional programming languages using minimal and elementary effort. The interpreter takes as input the result of the transformation mentioned above. The implementation of the interpreter is considerably shorter than that of byte-code based interpreters like Helium, Hugs and GHCi with a better performance. The better performance of the interpreter can be attributed to the simplicity of the intermediate formalism enabling a high-level abstract machine having large atomic actions with minimal interpretation overhead.

The structure of this paper is as follows. In Section 3.2 we introduce a new encoding of algebraic data types by functions and we compare this encoding with two existing encodings. In Section 3.3 we introduce the intermediate functional programming language Sapl. Sapl has, besides integers and their operations, no data types. Sapl is similar to the pure functional kernel of languages like Haskell and Clean. We show how to transform complex pattern-based function definitions to Sapl based on the representation of data types from Section 3.2. In Section 3.4 we define an interpreter for this language based on straightforward graph-rewriting techniques. We show how the interpreter can be optimised by using two simple annotations that can be added to Sapl programs. The performance of the optimised interpreter is compared with other implementations in Section 3.5. In Section 3.6 we give some conclusions and discuss further research possibilities.

3.2 Representation of Data Types by functions

In the lambda calculus several representations of algebraic data types by functions (or lambda expressions) exist. In this section we introduce a new representation and compare it with the two most important existing representations. We use two examples to demonstrate the differences: the Peano representation of natural numbers with the addition and predecessor operations and lists with the length and tail operations. We use Haskell syntax for all definitions, although some functions cannot be typed.

3.2.1 A New Representation of Data Types by Functions

Consider the following algebraic data type definition in Haskell or Clean:

$$typename\ t_1\ ..\ t_k\ ::=\ C_1\ t_{1,1}\ ..\ t_{1,n_1}\ |\ ..\ |\ C_m\ t_{m,1}\ ..\ t_{m,n_m}$$

We map this type definition with m constructors to m functions:

$$\begin{aligned} C_1\ v_{1,1}\ ..\ v_{1,n_1} &= \lambda f_1\ ..\ f_m\ \rightarrow\ f_1\ v_{1,1}\ ..\ v_{1,n_1} \\ .. & \\ C_m\ v_{m,1}\ ..\ v_{m,n_m} &= \lambda f_1\ ..\ f_m\ \rightarrow\ f_m\ v_{m,1}\ ..\ v_{m,n_m} \end{aligned}$$

Each constructor is represented by a function with the same name. Now consider the Haskell (multi-case) function f with as argument an element of this data type:

$$\begin{aligned} f\ (C_1\ v_{1,1}\ ..\ v_{1,n_1}) &= body_1 \\ .. & \\ f\ (C_m\ v_{m,1}\ ..\ v_{m,n_m}) &= body_m \end{aligned}$$

This function is converted to the following function without patterns:

$$\begin{aligned} f\ el\ =\ & el \\ & (\lambda\ v_{1,1}\ ..\ v_{1,n_1}\ \rightarrow\ body_1) \\ & .. \\ & (\lambda\ v_{m,1}\ ..\ v_{m,n_m}\ \rightarrow\ body_m) \end{aligned}$$

The body of each case is turned into a lambda expression that is placed as an argument of the data type element. The actual data type argument will select the correct lambda expression and apply it to the arguments of the constructor. Therefore we call a function corresponding to a constructor a *selector* function. The result of the transformation of recursive functions on recursive data types cannot be typed by Hindley-Milner type inference (see examples in the next section). This is not a problem because the functions can be typed before the transformation.

3.2.2 Examples

The Haskell definitions for the examples are (note that we defined *tail Nil* as *Nil* and *pred Zero* as *Zero* in order to have total functions):

$$
\begin{array}{lcl}
\mathit{data\ Nat} & = & \mathit{Zero} \mid \mathit{Suc\ Nat} \\
\mathit{add\ n\ Zero} & = & n \\
\mathit{add\ n\ (Suc\ m)} & = & \mathit{Suc\ (add\ n\ m)} \\
\mathit{pred\ Zero} & = & \mathit{Zero} \\
\mathit{pred\ (Suc\ n)} & = & n
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{data\ List\ t} & = & \mathit{Nil} \mid \mathit{Cons\ t\ (List\ t)} \\
\mathit{length\ Nil} & = & 0 \\
\mathit{length\ (Cons\ x\ xs)} & = & 1\ +\ \mathit{length\ xs} \\
\mathit{tail\ Nil} & = & \mathit{Nil} \\
\mathit{tail\ (Cons\ x\ xs)} & = & \mathit{xs}
\end{array}
$$

Using the transformation to functions this becomes:

$$
\begin{array}{lcl}
\mathit{Zero} & = & \lambda f\ g\ \rightarrow\ f \\
\mathit{Suc\ n} & = & \lambda f\ g\ \rightarrow\ g\ n \\
\mathit{add\ n\ m} & = & m\ n\ (\lambda\, \mathit{pm}\ \rightarrow\ \mathit{Suc\ (add\ n\ pm)}) \\
\mathit{pred\ n} & = & n\ \mathit{Zero}\ (\lambda\, \mathit{pn}\ \rightarrow\ n)
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{Nil} & = & \lambda f\ g\ \rightarrow\ f \\
\mathit{Cons\ x\ xs} & = & \lambda f\ g\ \rightarrow\ g\ x\ \mathit{xs} \\
\mathit{length\ ys} & = & \mathit{ys}\ 0\ (\lambda\, x\ \mathit{xs}\ \rightarrow\ 1\ +\ \mathit{length\ xs}) \\
\mathit{tail\ ys} & = & \mathit{ys\ Nil}\ (\lambda\, x\ \mathit{xs}\ \rightarrow\ \mathit{xs})
\end{array}
$$

pred and *tail* both have complexity *O(1)*. The functions *Zero, Suc, Nil, Cons, pred* and *tail* can be typed, but *add* and *length* cannot be typed using Hindley-Milner type inferencing. In general, the encoding of recursive functions on recursive data types cannot be typed. The definitions of *add* and *length* are explicitly recursive. In general, to encode recursive functions over recursive data structures, we need explicit recursion. This is not a problem since we use named functions instead of lambda expressions in our encoding. The notation is easy to read and close to the original Haskell data type and function definitions.

3.2.3 Church Encoding

For this encoding we need pairs with the selection functions *fst* and *snd*. They can be represented by functions as follows:

$$
\begin{array}{lcl}
\mathit{pair\ x\ y} & = & \lambda f\ \rightarrow\ f\ x\ y \\
\mathit{fst\ p} & = & p\ (\lambda\, x\ y\ \rightarrow\ x) \\
\mathit{snd\ p} & = & p\ (\lambda\, x\ y\ \rightarrow\ y)
\end{array}
$$

The Church encoding is a generalization of the Church numerals. The representation described here is based on Berarducci and Bohm [BB93] and Barendregt [Bar97]. For comparison reasons we use a slightly different notation than is generally used for describing Church numerals:

$$
\begin{array}{lcl}
\mathit{Zero} & = & \lambda f\ g\ \rightarrow\ f \\
\mathit{Suc}\ n & = & \lambda f\ g\ \rightarrow\ g\ (n\ f\ g) \\
\mathit{add}\ n\ m & = & m\ n\ (\lambda\ \mathit{rpm}\ \rightarrow\ \mathit{Suc}\ \mathit{rpm}) \\
\mathit{pred}\ n & = & \mathit{snd}\ (n\ (\mathit{pair}\ \mathit{Zero}\ \mathit{Zero})\ (\lambda\ p\ \rightarrow\ \mathit{pair}\ (\mathit{Suc}\ (\mathit{fst}\ p))\ (\mathit{fst}\ p)))
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{Nil} & = & \lambda f\ g\ \rightarrow\ f \\
\mathit{Cons}\ x\ xs & = & \lambda f\ g\ \rightarrow\ g\ x\ (xs\ f\ g) \\
\mathit{length}\ ys & = & ys\ 0\ (\lambda\ x\ \mathit{rxs}\ \rightarrow\ 1\ +\ \mathit{rxs}) \\
\mathit{tail}\ xs & = & \mathit{snd}\ (xs\ (\mathit{pair}\ \mathit{Nil}\ \mathit{Nil}) \\
 & & \quad (\lambda\ x\ \mathit{pxs}\ \rightarrow\ \mathit{pair}\ (\mathit{Cons}\ x\ (\mathit{fst}\ \mathit{pxs}))\ (\mathit{fst}\ \mathit{pxs})))
\end{array}
$$

In the *add* definition *add n (Suc m)* can be defined using the result of *add n m* (represented by *rpm*). The same holds for *length*. But in predecessor *pred (Suc n)* cannot be expressed in terms of *pred n*. Instead we need access to *n* in *Suc n* (we need to destruct *Suc n*). Kleene ([Bar84]) found a way to overcome this by the use of pairs. In such a pair *n* is combined with the result of the recursive call, so access to *n* is also possible. For *tail* we also need this pair construction. Through this construction *pred n* has complexity *O(n)* and *tail xs* has complexity *O(length xs)*. In this encoding the recursion is put into the data structures. Therefore, functions on data structures do not have to be recursive themselves. A disadvantage is that this encoding only works fine for iterative and primitive recursive functions (see [BB85]). For destructor functions we need the pair construction. In the Church encoding data types and functions acting on them can be typed using Hindley-Milner type inference.

3.2.4 Representation according to Berarducci and Bohm

Another representation is described in Berarducci and Bohm [BB85] and Barendregt [Bar97]. Again we adapted the notation to make a comparison with the other representations possible.

$$
\begin{array}{lcl}
\mathit{Zero} & = & \lambda f\ g\ \rightarrow\ f\ f\ g \\
\mathit{Suc}\ n & = & \lambda f\ g\ \rightarrow\ g\ n\ f\ g \\
\mathit{add}\ n\ m & = & m\ (\lambda\ fz\ fs\ \rightarrow\ n)\ (\lambda\ \mathit{pm}\ fz\ fs\ \rightarrow\ \mathit{Suc}\ (\mathit{pm}\ fz\ fs)) \\
\mathit{pred}\ n & = & n\ (\lambda\ fz\ fs\ \rightarrow\ \mathit{Zero})\ (\lambda\ \mathit{pn}\ fz\ fs\ \rightarrow\ \mathit{pn})
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{Nil} & = & \lambda f\ g\ \rightarrow\ f\ f\ g \\
\mathit{Cons}\ x\ xs & = & \lambda f\ g\ \rightarrow\ g\ x\ xs\ f\ g \\
\mathit{length}\ ys & = & ys\ (\lambda\ \mathit{fn}\ \mathit{fc}\ \rightarrow\ 0)\ (\lambda\ x\ xs\ \mathit{fn}\ \mathit{fc}\ \rightarrow\ 1\ +\ xs\ \mathit{fn}\ \mathit{fc}) \\
\mathit{tail}\ ys & = & ys\ (\lambda\ \mathit{fn}\ \mathit{fc}\ \rightarrow\ \mathit{Nil})\ (\lambda\ x\ xs\ \mathit{fn}\ \mathit{fc}\ \rightarrow\ xs)
\end{array}
$$

The basic idea in this representation is that the functions handling the different cases are propagated by the functions representing the data structures. Therefore, functions on data structures do not have to be recursive themselves. Here *pred n* and *tail xs* have complexity *O(1)*. In general, destructor functions have complexity $O(1)$, making this representation more powerful than the Church encoding. In this

representation *Zero, Suc, Nil* and *Cons*, as well as the functions acting on them cannot be typed by Hindley-Milner type inference.

3.2.5 Conclusions

Our representation is more efficient than the Church encoding, because it realizes destructor functions with $O(1)$. Although this also holds for the representation of Berarducci and Bohm, the use of named functions and explicit recursion in our representation result in a simpler representation, which is suitable for an efficient implementation (see Section 3.4).

3.3 SapI: An intermediate Functional Language

SapI is an intermediate language that can be used for the compilation and interpretation of functional programming languages like Haskell and Clean. The main difference between SapI and the intermediate formalisms normally used is the absence of algebraic data types and constructs for pattern matching in SapI. This makes SapI a compact and simple language. In Section 3.4 we show that it is possible to make an efficient implementation for SapI. SapI is described by the following syntax:

$$
\begin{array}{lll}
\mathit{function} & ::= & \mathit{identifier}\ \{\mathit{identifier}\} * \ '='\ \ \mathit{expr} \\
\mathit{expr} & ::= & \mathit{application} \mid \ '\lambda'\ \{\mathit{identifier}\} + \ '\rightarrow'\ \mathit{expr} \\
\mathit{application} & ::= & \mathit{factor}\ \{\mathit{factor}\}* \\
\mathit{factor} & ::= & \mathit{identifier} \mid \mathit{integer} \mid \ '('\ \mathit{expr}\ ')'
\end{array}
$$

A function has a name followed by zero or more variable names. An expression is either an application or a lambda expression. In an expression only variable names, integers and other function names may occur. SapI function definitions start in the first column and can extend over several lines (as long as these are indented). SapI is un-typed. The language has the usual lazy rewrite semantics (see Section 3.4). For efficiency we added integers and their basic operations to the language. In SapI it is common that a curried application of a function is the result of a computation. This result will be presented as the application of the function name to the evaluated arguments.

SapI's main difference with the lambda calculus is the use of explicitly named functions (enabling explicit recursion) which makes SapI usable as a basic functional programming language and suitable for an efficient implementation.

For the use of SapI as an intermediate language for implementing lazy functional languages like Haskell and Clean we must translate constructs from these languages to SapI functions. Constructions like list-comprehensions, *where* and *let(rec)* expressions can be converted to functions with standard techniques as described in [PJ87] and [PvE93]. Algebraic data types and simple pattern-based functions are treated specially using the translation scheme from Section 3.2. In the next subsection the transformation of complex pattern-based functions is sketched.

3.3.1 Compiling Complex Pattern Definitions to Functions

In the implementations of Haskell and Clean pattern-based definitions are traditionally compiled to dedicated structures in a special pattern formalism that can be used to generate pattern-matching code (Augustsson [Aug85] and Peyton Jones [PJ87]). Here we transform a pattern-based function definition from Clean or Haskell to a single SapI function without patterns. This function is capable of handling an actual call for the original pattern-based function. The conversion to a single function can be obtained using techniques similar to those used for the generation of pattern-matching code (see [Aug85] and [PJ87]). We use three examples to illustrate this conversion: *mappair* (*zipWith*), *samelength* and *complex*. Note that the pattern compiler introduces a name for every constructor (e.g. *as* in *mappair*) and uses existing names whenever possible (e.g. *ps* and *qs* in *samelength*).

$$
\begin{array}{lll}
mappair\ f\ Nil & zs & = Nil \\
mappair\ f\ (Cons\ x\ xs) & Nil & = Nil \\
mappair\ f\ (Cons\ x\ xs) & (Cons\ y\ ys) & = Cons\ (f\ x\ y)\ (mappair\ f\ xs\ ys)
\end{array}
$$

$$
\begin{array}{lll}
samelength\ Nil & Nil & = True \\
samelength\ (Cons\ x\ xs) & (Cons\ y\ ys) & = samelength\ xs\ ys \\
samelength\ ps & qs & = False
\end{array}
$$

$$
\begin{array}{ll}
complex\ (Cons\ a\ (Cons\ b\ (Cons\ c\ Nil))) & = a + b + c \\
complex\ (Cons\ a\ (Cons\ b\ Nil)) & = 2 * a + b \\
complex\ (Cons\ a\ Nil) & = 3 * a \\
complex\ xs & = 0
\end{array}
$$

The translation to SapI results in:

$$
\begin{array}{l}
mappair\ f\ as\ zs\ =\ as\ Nil\ (\lambda\, x\ xs\ \rightarrow\ zs\ Nil\ (\lambda\, y\ ys\ \rightarrow \\
\qquad\qquad Cons\ (f\ x\ y)\ (mappair\ f\ xs\ ys)))
\end{array}
$$

$$
\begin{array}{l}
samelength\ ps\ qs\ =\ ps\ (qs\ True\ (\lambda\, y\ ys\ \rightarrow\ False)) \\
\qquad\qquad (\lambda\, x\ xs\ \rightarrow\ qs\ False\ (\lambda\, y\ ys\ \rightarrow samelength\ xs\ ys))
\end{array}
$$

$$
\begin{array}{l}
complex\ xs\ =\ xs\ 0\ (\lambda\ a\ p1\ \rightarrow\ p1\ (mult\ 3\ a)\ (\lambda\, b\ p2\ \rightarrow \\
\qquad\qquad p2\quad (add\ (mult\ 2\ a)\ b) \\
\qquad\qquad\qquad (\lambda\, c\ p3\ \rightarrow\ p3\ (add\ (add\ a\ b)\ c)\ (\lambda\, p4\ p5\ \rightarrow 0))))
\end{array}
$$

3.4 An Interpreter for SapI

The only operations in SapI programs are function application and a number of (built-in) integer operations. Therefore an interpreter can be kept small and elegant.

The interpreter is implemented in C and is based on straightforward graph-reduction techniques as described in Peyton Jones [PJ87, PJL92], Plasmeijer and van Eekelen [PvE93] and Kluge [Klu04]. We assume that a pre-compiler has eliminated all algebraic data types and pattern definitions (as described earlier) and all let(rec)- and where- clauses and lifted all lambda expressions to the global level. The interpreter is only capable of executing function rewriting and the basic operations on integers. The most important features of the interpreter are:

- It uses 4 types of memory Cells. A Cell corresponds to a node in the syntax tree and is either an: Integer, (Binary) Application, Variable or Function Call. To keep memory management simple, all Cells have the same size. A type byte in the Cell distinguishes between the different types. Each Cell uses 12 bytes of memory.

- The memory heap consists only of Cells. The heap has a fixed size, definable at start-up. We use a mark and (implicit) sweep garbage collection. Cells are not recollected, but the dirty bit is inverted after every mark.

- It uses a single argument stack containing only references to Cells. The C (function) stack is used as the dump for keeping intermediate results when evaluating strict functions (numeric operations only) and for administration overhead during the marking phase of garbage collection.

- The state of the interpreter consists of the stack, the heap, the dump, an array of function definitions and a reference to the node to be evaluated next. In each state the next step to be taken depends on the type of the current node: either an application node or a function node.

- It reduces an expression to head-normal-form. The printing routine causes further reduction. This is only necessary for arguments of curried functions.

The interpreter is based on the following 'executable specification' (without integers and their operations):

$$data\ Expr\ =\ App\ Expr\ Expr \mid Func\ Int\ Int \mid Var\ Int$$

The first *Int* in *Func Int Int* denotes the number of arguments of the function, the second *Int* the position of the function definition in the list of definitions. The *Int* in *Var Int* indicates the position on the stack where the argument can be found.

The interpreter consists of three functions:

$$\begin{array}{lcl} instantiate\ (App\ l\ r)\ es & = & App \quad (instantiate\ l\ es)\ (instantiate\ r\ es) \\ instantiate\ (Var\ n)\ es & = & es\ !!\ n \\ instantiate\ x\ es & = & x \\ \\ rebuild\ e\ [] & = & e \\ rebuild\ e\ (x\ :\ xs) & = & rebuild(App\ e\ x)\ xs \end{array}$$

$$
\begin{array}{lll}
eval :: & Expr \rightarrow [Expr] \rightarrow [Expr] \rightarrow Expr \\
eval & (App\ l\ r)\ es\ fs = eval\ l\ (r : es)\ fs \\
eval & (Func\ na\ fn)\ es\ fs \\
= & \textbf{if} \quad length\ es \geq na \\
& \textbf{then}\ eval\ (instantiate\ (fs\ !!\ fn)\ es)\ (drop\ na\ es)\ fs \\
& \textbf{else}\ rebuild\ (Func\ na\ fn)\ es
\end{array}
$$

Here *es* represents the stack and *fs* the list of function body definitions. One of the benchmarks in Section 3.5 is a SapI version of the interpreter (including integers and their operations), which is the translation to SapI of the Haskell version of the interpreter (a meta-circular implementation for SapI). The C versions (including integers and operations on them) of *eval* and *instantiate* are straightforward implementations of this specification and fit on less than one page.

3.4.1 Optimising the SapI Interpreter

For data-type-free programs the interpreter from the previous subsection has a performance comparable to Helium, GHCi and Amanda. But for programs involving algebraic data types the performance is worse. The difference depends on the number of alternatives and the complexity of the data type definition and varies from 30% slower for programs involving only if-then-else constructs, to several hundreds of times slower for programs involving complex data types and pattern matching (see section 3.5). This is not surprising because a pattern definition is converted to one large function containing all different cases. Instantiation of such a function is therefore relatively expensive, particularly because only a small part of the body will actually be used in a call for the function.

For optimising the SapI interpreter we used both general optimisation techniques, commonly used for implementing functional languages, as well as techniques that are more specific for the way SapI handles data types and pattern definitions.

General Optimisations

We use a more efficient memory representation for function calls with one or two arguments. For these function applications *APP* nodes are removed. This reduces the size of the bodies of functions and consequently copying overhead.

In the interpreter curried function calls are rebuilt. This can be prevented by keeping a reference to the top node of the application. If the number of arguments for a function call can be computed at compile time, the top node of a curried call can be marked. In this way an attempt to reduce a curried call can even be prevented.

Applying these two optimisations results in an average speed-up of 60% (see section 3.5). This speed-up is high since many functions have only 1 or 2 arguments and because SapI programs contain many curried functions (due to the representation of data types by functions).

Specific Optimisations

We applied two specific optimisations. The first one addresses the instantiation problem for functions that are the result of the translation of pattern-based function definitions. The second one optimises the use of lambda expressions in these functions. Although the speed-up realized by these optimisations is significant, the implementation of them requires only small changes in the interpreter.

Selective Instantiation of Function Bodies The body of a transformed pattern-based definition consists of the application of a so-called selector function (see Section 3.3) to a number of arguments consisting of anonymous local function definitions. The selector function will select one of these local function definitions and apply it to the arguments of the corresponding constructor. All other arguments of the selector function will be ignored. In the *mappair* example below we have tagged the applications of selector functions with the keyword *select*.

$$
\begin{array}{l}
mappair\ f\ as\ zs\ = \\
\qquad\qquad select\ as\ Nil\ (\lambda\, x\ xs\ = \\
\qquad\qquad select\ zs\ Nil\ (\lambda\, y\ ys\ =\ Cons\ (f\ x\ y)\ (mappair\ f\ xs\ ys)))
\end{array}
$$

The interpreter uses the *select* (semantically equivalent to the identity function) tag to optimise the instantiation of the body of *mappair*. Instead of copying the entire body, at first only the selector function part is instantiated (*as*) and depending on the result (*Nil* or *Cons x xs*), the correct remainder is instantiated. This is similar to evaluating the condition of an *if* expression before we decide to build the *then* part or the *else* part (but not both). In fact, in Sapl *True* and *False* are also implemented as selector functions. The optimisation is applied recursively to the bodies of all local definitions.

The optimisation realised in this way is significant. Varying from 30% faster for programs involving only if-then-else constructs, to up to 500 times faster for programs involving complex data type definitions like interpreters etc.

We can add the *select* tag during the transformation of the pattern-based function definition to Sapl, but it is also possible to infer the application of selector functions by a compile time analysis of a Sapl program. Selector functions must be recognized and the propagation of arguments and results of functions that are selector functions must be inferred. In this way this optimisation is a generic one and can even be used for the efficient reduction of lambda expressions.

Inlining of Local Definitions As a last optimisation we again consider the bodies of transformed pattern-based definitions. They contain local function definitions corresponding to the different cases. Normally these definitions are lambda lifted to the global level. During this lifting extra arguments are added to the function, causing extra stack operations at run-time. These local functions can also be reduced in the context of the reduction of the surrounding function call. This means that the local function is called (reduced) while the arguments of the main function are still on the stack and that at the end all arguments together are cleared from the

stack. This can only be done because the reduction to head-normal-form of the local function call is necessary for the reduction to head-normal-form of the original function call, which is indeed the case for these transformed pattern-based functions. This optimisation results in an extra speed-up of about 10 to 25% for programs involving transformed pattern-based functions (see section 3.5). The optimisation is implemented by replacing $\rightarrow$ by $=$ in the local definition as a signal for the interpreter not to lambda lift this local function (see example in 3.4.1).

Again this optimisation can be applied not only for local definitions in translated pattern-based functions, but for all local function calls that must be reduced to head-normal-form while reducing the surrounding function call. But the gain for SapI programs will be higher than for applying this optimisation for other functional languages, because SapI programs, due to the translation scheme for pattern-based functions, contain more local function definitions.

3.5 Benchmarks

In this section we present the results of several benchmark tests for SapI and a comparison of SapI with other implementations. We ran the benchmarks on a 2.66 Ghz Pentium 4 computer with 512Mb of memory under Windows XP. SapI was implemented using the Microsoft Visual C++ compiler using the -O2 option. The benchmark programs we used for the comparison are:

1. **Prime Sieve** The prime number sieve program, calculating the 5000th prime number.
2. **Symbolic Primes** Symbolic prime number sieve using Peano numbers, calculating the 280th prime number.
3. **Interpreter** An interpreter for SapI, as described in Section 3.4 (including integers). As an example we coded the prime number sieve for this interpreter and calculated the 100th prime number.
4. **Fibonacci** The (naive) Fibonacci function, calculating *fib 35*.
5. **Match** Nested pattern matching (5 levels deep) like the *complex* function from section 3.3.1, repeated 2000000 times.
6. **Hamming** The generation of the list of Hamming numbers (a cyclic definition) and taking the 1000th Hamming number, repeated 4000 times.
7. **Twice** A higher order function (*twice twice twice twice (add 1) 0*), repeated 400 times.
8. **Sorting** Tree Sort (6000 elements), Quick Sort (6000 elements), Merge Sort (40000 elements, merge sort is much faster) and Insertion Sort (6000 elements).
9. **Queens** Number of placements of 11 Queens on a 11 * 11 chess board.
10. **Knights** Finding a Knights tour on a 5 * 5 chess board.
11. **Parser Combinators** A parser for Prolog programs based on Parser Combinators parsing a 17000 lines Prolog program.

Table 3.1: SAPL with/without Selective Instantiation (Time in seconds)

	Pri	Sym	Inter	Fib	Match	Twi	Sort	Qns	Kns	Parse	Plog
With	11.4	6.0	2.2	11.6	14.7	11.0	1.0	10.5	4.0	8.0	0.2
Without	21.5	107.0	53.0	19.2	23.0	10.9	17.8	16.0	6.1	16.0	106.0

12. **Prolog** A small Prolog interpreter based on unification only (no arithmetic operations), calculating ancestors in a four generation family tree, repeated 500 times.

For sorting a list of size n we used a source list consisting of numbers 1 to n. The elements that are 0 modulo 10 are put before those that are 1 modulo 10, etc.

Three of the benchmarks (*Interpreter*, *Prolog* and *Parser Combinators*) are realistic programs, the others are typical benchmark programs that are often used for comparing implementations. They cover a wide range of aspects of functional programming (lists, laziness, deep recursion, higher order functions, cyclic definitions, pattern matching, heavy calculations, heavy memory usage). All times are machine measured. The programs where chosen in such a way that they ran for at least several seconds (interpreters only). Therefore start-up times can be neglected. The output was always converted to a single number (e.g. by summing the elements of a list) to eliminate the influence of slow output routines.

The input for the Sapl interpreter is code generated by an experimental data type and pattern compiler from sources equivalent to the Haskell and Clean programs (only minor syntactic differences). This compiler also generates the annotations needed for the optimisations. The *inline* optimisation is only applied for the lambda expressions that are the result of encoding a pattern- based definition. The benchmarks programs can be found in [SAP].

3.5.1 Optimisations for Sapl

In table 3.1 we first compare Sapl with and without the selective instantiation optimisation. In this comparison the other optimisation are not applied. *Hamming* is missing because the version of the interpreter without selective instantiation does not support cyclic definitions. We conclude that the selective instantiation optimisation is essential. Because Sapl also uses selective instantiation to optimise the if- then-else construct there is a speed-up for all benchmarks except *twice* (the only benchmarks without if-then-else and data structures). In the other examples the speed-up varies from around 1.5 times (*Primes, Fibonacci, Match, Queens, Knights*), around 20 times (*Symbolic Primes, Interpreter, Sorting*) to more than 500 times for *Prolog* (due to the complicated *unification* function).

Table 3.2 shows the results of applying the other optimisations.

- **Full** The fully optimised interpreter (Select, Mem and Inline).
- **Select** The interpreter using only the selective instantiation optimisation.

Table 3.2: Comparison Versions of SAPL (Time in seconds)

	Pri	Sym	Inter	Fib	Match	Ham	Twi	Sort	Qns	Kns	Parse	Plog
Full	6.1	17.6	7.8	7.3	8.5	6.4	7.9	5.9	6.5	2.0	4.4	4.7
Select	11.4	37.6	14.3	11.6	14.7	11.3	11.0	9.4	10.6	4.0	8.0	10.4
Mem	6.2	28.0	9.3	7.5	9.0	8.0	7.9	6.4	7.0	2.7	4.9	6.7
Inline	11.4	24.4	12.9	11.5	14.4	9.2	11.0	8.7	10.0	3.3	7.5	7.8

Table 3.3: Different Memory Configurations (Time sec, Heap/Stack kB)

	Pri	Sym	Inter	Fib	Match	Ham	Twi	Sort	Qns	Kns	Parse	Plog
Heap	223	47	2350	12	101	105	785	2350	43	18	9700	150
Stack	270	35	1100	1	1	1	1	200	1	1	200	4
10.8 Mb												
Time	6.7	17.1	13.0	8.0	9.2	6.9	9.1	6.7	7.0	2.1	17.0	5.2
% GC	15	12	46	13	18	14	18	21	17	14	76	17
nr GCs	87	204	150	117	157	100	120	83	114	32	190	83
24 Mb												
Time	6.4	17.5	8.8	7.8	9.1	6.7	8.8	6.5	7.0	2.1	6.0	5.1
% GC	13	10	24	13	18	15	15	15	14	14	38	16
nr GCs	38	91	61	53	70	45	52	37	51	15	40	37
60 Mb												
Time	6.4	18.6	8.3	7.6	9.1	6.6	8.5	6.5	6.9	2.1	5.0	5.1
% GC	13	10	18	13	16	15	13	16	14	14	24	16
nr GCs	15	36	24	28	28	18	21	15	21	6	14	15

- **Mem** The interpreter using selective instantiation and the efficient representation of functions with 1 or 2 arguments.
- **Inline** The interpreter using selective instantiation and inlining of lambda expressions in encoded pattern-based functions.

From this comparison we learn that the fully optimised version is about 1.8 times faster than the version using only selective instantiation, 1.2 times faster than the version with selective instantiation and memory optimisation and 1.6 times faster than the version with selective instantiation and inlining. The benefit from the inline optimisation is modest, but the implementation of it in the run-time system consists of only moving a stack pop operation to another line. The more efficient memory representation gives a significant speed-up.

In table 3.3 we compare the behavior of SAPL for a number of memory configurations: 10.8 Mb (90000 Cells), 24 Mb (2000000 Cells) and 60 Mb (5000000 Cells). 900000 Cells is the minimal heap size needed to run all benchmarks. We also give peak heap and stack usage in Kb and percentage of time spent in GC and number of GCs. Because heap and stack usage are only measured at GC the actual maximum values can be (slightly) higher than those measured. For these tests we used a garbage collector with an explicit sweep phase instead of the implicit sweep (during memory allocation). This is done to make it possible to give meaningful figures about time spent in garbage collection. The price to be paid is a small performance

Table 3.4: Run-Times (in seconds) for different Implementations

	Pri	Sym	Inter	Fib	Match	Ham	Twi	Sort	Qns	Kns	Parse	Plog
SAPL	6.1	17.6	7.8	7.3	8.5	6.4	7.9	5.9	6.5	2.0	4.4	4.7
Helium	13,6	17,6	16,3	12,2	17.4	12.8	23.2	10,4	9,7	3.4	8.4	7.1
Amanda	18.0	33.0	-	8.8	17.2	14.0	-	12.5	7.7	2.4	10.9	8.5
GHCi	18.0	19.5	25.0	38.6	35.3	23.5	19.3	13.8	24.0	7.0	8.7	11.9
Hugs	44.0	26.0	-	120.0	66.0	36.0	-	54.0	42.0	13.0	10.4	16.2
GHC	1.8	1.5	8.2	4.0	4,1	3.8	6.6	1.6	3.7	0.9	2.3	1.3
GHC -O	0.9	1.5	1.8	0.2	1.0	1.4	0.1	1.1	0.4	0.2	1.6	0.4
Clean	0.9	0.8	0.6	0.2	0.9	1.4	2.4	0.7	0.4	0.2	4.9	0.6

penalty ($< 10\%$) and the use of an administration array for the collected free cells.

We conclude that if the peak heap memory stays under 30% of the total heap size execution times do not differ too much. If peak heap usage rises above 50% of total memory, performance drops radically and the amount of time spent in garbage collection grows rapidly. Because SapI has a fixed heap, the memory management overhead is lower than in implementations with a flexible heap. SapI uses relatively few GC cycles, because SapI has a fixed heap and only starts garbage collection if there are less than 1000 free cells left.

The stack usage of SapI is modest. Note, however, that SapI also uses the C stack. The maximum amount of C stack for SapI is 8Mb.

3.5.2 Comparison with other Implementations

In this subsection we compare SapI with several other interpreters: Amanda V2.03 [Brub], Helium 1.5 [Sof], Hugs 20050113 [Hug] and GHCi V6.4 [GHC] and with the GHC V6.4 and Clean V2.1 compilers. We used the same amount of (fixed or maximal) heap space (64 Mb) and stack space (8 Mb) for all examples whenever this was possible (for Amanda the stack size cannot be set). For *Interpreter* and *Twice* the Amanda results are missing because of a stack overflow. Hugs also could not run these examples (C stack overflow).

Run-Time Comparison

The run-time results can be found in table 3.4. The results show us that the SapI interpreter is almost 2 times faster than Amanda and Helium, about 3 times faster than GHCi and between 1.5 and 15 times faster than Hugs.

For the compilers there is more variation in the results due to the different optimisations applied by them. Comparing SapI with GHC, the average speed-up of GHC is less than 3 times. The speed-ups of GHC -O and Clean vary between 1.1 (*Parser Combinators* in Clean) and 80 (*Twice* in GHC -O).

Table 3.5: Comparison Max Heap (kB) usage (upper) and GC time (%) (lower)

	Pri	Sym	Inter	Fib	Mch	Ham	Twi	Sort	Qns	Kns	Parse	Plog
SAPL	223	47	2344	12	101	107	762	2344	43	17	9700	150
Helium	774	16000	3000	258	774	516	1800	9000	258	256	10700	500
GHC	140	21	1800	6	46	50	800	1600	7	6	7000	50
SAPL	13	10	24	13	18	15	15	15	14	14	38	16
Helium	47	7	45	5	25	25	59	7	12	46	47	17
GHC def	18	1	87	1	22	16	67	5	1	45	70	25
GHC 24M	1	1	23	1	1	1	4	1	1	5	59	1

Comparison of Heap Usage

In table 3.5 we compare the memory usage and the time spent in garbage collection of SapI (24 Mb heap) with that of Helium (standard heap) and the GHC compiler (standard and 24Mb initial heap). For Hugs, GHCi and Amanda no meaningful figures about memory usage can be given. We do not include a stack size comparison because SapI also uses an unknown part of the C stack.

We conclude that GHC and SapI use roughly the same amount of heap but that Helium uses more heap. The difference between SapI and GHC can be explained by the fixed Cell size of 12 bytes used by SapI. The unexpected high value of Helium for *Symbolic Primes* is probably a memory leak.

The amount of time spent in garbage collection of SapI is mostly slightly lower than that of Helium and lower than that of GHC (default heap) for memory intensive programs like *Interpreter* and *Parser*. Variations of the (initial) heap size have only a small effect on the SapI and Helium performance, but have a big impact on the performance of GHC. Setting the initial heap to 24Mb gives an almost 3-time speed-up for *Interpreter* and *Twice*, but halves the speed of almost all other benchmarks.

3.5.3 Discussion about Interpreter Comparison

What is the source of the good performance of SapI compared with GHCi, Helium, Hugs and Amanda? The simplified memory management contributes to this better performance, but cannot be the only source (see table 3.5). Helium performs an overflow check on integer operations, which slows down integer intensive programs. If we compare SapI with Amanda we see that for (almost) data type free programs there is not much difference in performance (*Fibonacci*, *Queens* and *Knights*). The difference in performance appears for programs using data types and pattern matching. Amanda uses a similar implementation of graph reduction as SapI, but has a less sophisticated implementation of pattern matching using case-by-case matching [Brua]. If we compare the performance of SapI with that of GHCi, Helium and Hugs we see that SapI already has a better performance for data type free programs (*Twice, Fibonacci*). This increase in speed remains about the same for programs using data types and pattern matching. Helium uses techniques based on the STG machine to generate LVM byte code [Lei03]. This byte code is interpreted. GHCi also compiles to byte code and is based on the GHC compiler that also uses the STG

machine [PJ92]. The Hugs implementation is based on byte code interpretation too. The SapI interpreter is based on graph rewriting only and has no special constructs for data types and pattern matching. This enables a simple, high- level abstract machine with few, relatively large, atomic operations. There is no need for a more low level intermediate (byte code) formalism. The main difference between an interpreter and a compiler is that an interpreter has to check what to do next at every step. Keeping this overhead as small as possible is important for the construction of efficient interpreters. The easiest way to keep this overhead small is to use large atomic steps in the interpreter. Byte code instructions are mostly quite small. SapI has a simple structure and uses large atomic steps. As a result the interpretation overhead for SapI is lower than that for byte code based interpreters. The atomic operations in the SapI interpreter are:

- Push a reference on the stack.
- Instantiate a function body, clear its arguments from the stack and place the result at the top application node.
- Call a built-in function, clear arguments from stack and place result at top application node.
- For a function call with as body a selector function application: Partly instantiate the body, recursively call *eval* for this instantiation and use the result to select and instantiate the appropriate other part of the body.

Except for the *push* operation these are all relatively large operations. The only benchmark for which the SapI interpreter is not significant faster than Helium and GHCi, is *Symbolic Primes*. For this example the bodies of the (local) functions are mostly very small. Therefore the interpretation overhead will be much higher and comparable to the overhead of GHCi, Helium and Hugs.

Benefits of the Functional Encoding for the Interpreter Performance

First of all, we already concluded that the selective instantiation optimisation is essential for an efficient implementation of pattern-based function definitions using this encoding. It is therefore useless to try to run a SapI program using another interpreter or compiler that doesn't uses the selective instantiation optimisation. Furthermore, in the previous subsection we concluded that the extra efficiency of the SapI interpreter is not a result of the functional encoding and its implementation, but is a result of the simpler structure of the interpreter using a high level abstract machine with minimal interpretation overhead. The functional encoding enables this simple structure. It is possible to implement a traditional pattern matcher along the same lines as the functional pattern matcher with comparable performance, because both are based on the same techniques for encoding the pattern-based definition (see section 3.3.1).

We conclude that the most important benefit of the functional encoding is that it enables an elegant implementation of algebraic data types and pattern matching entirely within a pure functional domain and that this implementation can be made efficient by applying generic optimisations to a basic graph- rewriting interpreter.

3.6 Conclusions and further Research

In this paper we have defined the minimal (intermediate) functional programming language SapI and an interpreter for it, based on a new variant of the Church encoding for algebraic data types. SapI consists of pure functions only and has, besides integers, no other data types. For SapI we have achieved the following results:

- The representation of data structures as functions in SapI is more efficient than the Church encoding and the encoding of Berarducci and Bohm. The use of explicitly named functions (enabling explicit recursion) instead of lambda expressions enables an efficient implementation of this representation. We also showed how to translate pattern-based function definitions to SapI. This makes SapI usable as an intermediate language for interpretation of programs written in languages like Clean or Haskell.
- We described an efficient interpreter for SapI based on straightforward graph rewriting techniques. The basic version of the interpreter is an ideal subject for educational purposes and for experimenting with implementation issues for functional languages. After applying two optimisations to speed up the execution of functions that are the result of the translation of pattern-based function definitions, the interpreter turns out to be competitive in a comparison with other interpreters. The results show us that for interpretation a high-level abstract machine with large atomic operations yields better results than low-level byte code interpreters based on techniques used for compilers.

3.6.1 Future Work

We plan to investigate the following issues for SapI:

- We want to investigate whether the techniques used for implementing SapI are also usable for realizing a compiler. We did some small experiments for this. We hand compiled the internal SapI data structures to C code for a few benchmarks. This eliminates interpretation overhead and makes it possible to hard code the instantiation of functions (instead of a recursive copy). Speed-ups of 2 to 3 times seem possible, but more experiments are needed.
- We want to extend SapI with IO features for creating interactive programs. Because SapI is an interpreter it is also possible to use SapI only as a calculation engine for another environment that does the IO.
- We want to investigate applications of SapI. For example, SapI can be used at the client side of Internet browsers as a plug-in, or inside a spreadsheet application.

Chapter 4

From Interpretation to Compilation

[1] ***Abstract*** In this paper we sketch some experiments with the construction of a simple compiler for a high level intermediate lazy functional language, with C++ as target language. Because the compiler is intended for educational and experimental use, simplicity and clearness of construction are considered to be more important than efficiency. Starting point for the construction is a simple interpreter. In a first step this interpreter is turned into a simple compiler in a straightforward manner. The performance of a number of compiled benchmarks is analysed in a comparison with the interpreter and the Clean and GHC compilers. This analysis leads to some suggestions for optimisations. Of these optimisations tail recursion optimisation and optimisation of numerical functions and numerical (sub)expressions in functions are implemented. It turns out that in many cases these optimisations suffice to obtain a competitive performance.

4.1 Introduction

The construction of efficient compilers for lazy functional programming languages like Clean [PE01] and Haskell [PJ03] is a complex task. Compilers like GHC [GHC] and Clean are large complicated systems that are too complex for study in introductory courses on the implementation of functional programming languages. Therefore, there is a need for simple compilers for educational purposes. Our main goal is to give the reader some insight in what kind of optimisations are important for obtaining an efficient implementation of lazy functional languages.

In [JKP06] (chapter 3) we constructed a simple but efficient interpreter for the lazy functional language SapI. SapI can be used as an intermediate language for the interpretation of languages like Clean and Haskell. We already constructed a Clean to SapI translator. Several versions of the SapI interpreter exist. One of these versions is a Java applet implementation that can be loaded into Internet Browsers and which makes it possible to run Clean programs at the client side of Internet applications ([PAK07] and [PJKA08] (chapter 6)).

[1] Originally published as [JKP08a]

In this paper we investigate how we can extend the Sapl interpreter to a Sapl compiler with a reasonable performance. We use C++ as target language. The construction is made in two steps. In the first step we convert the interpreter into a straightforward but naive compiler. We then use a number of benchmarks to analyse the performance of the generated code in a comparison with the Clean and GHC compiler. It turns out that in some cases the performance is already quite good but that in other cases the performance is still very bad (more than 30 times slower). In an analysis of the characteristic of the poor performing benchmarks, it turns out that they often have some commonalities like the (heavy) use of tail recursive functions and the presence of many purely numeric functions or sub-expressions. Therefore, in the second step, we focus on improving the performance of the compiler by optimising tail recursions and numeric functions and sub-expressions. The resulting compiler is again compared with Clean and Haskell and the basic compiler using the same set of benchmarks. It turns out that the resulting performance is now acceptable in almost all cases.

Summarising, the contributions of this study are the stepwise construction of a simple compiler for a lazy (intermediate) functional programming language with the following characteristics:

- The compilers translates to concise and readable C++ functions (for a functional programmer knowing C++) that are in 1-1 correspondence with the original functions. The C++ functions give the programmer clear insight into how constructs from functional programming language are implemented.
- It gives the reader insight into what kind of optimisations are important for obtaining an efficient implementation of lazy functional languages.
- The user can easily add functions to the generated code and can modify generated functions to experiment with alternative optimisations.
- The performance of the resulting programs is in many cases competitive with that of Clean and Haskell.

The structure of this paper is as follows. In Section 4.2 we introduce the intermediate functional programming language Sapl. In Section 4.3 we sketch an interpreter for Sapl. This interpreter is the starting point for the construction of the compiler. The compiler is described in Section 4.4. We describe the compiler in a number of steps. First a basic version of the compiler is introduced that is a straightforward and simple extension of the interpreter. The performance of a set of benchmarks compiled with this compiler and the Clean and GHC compiler is used to make a comparison. The results of this comparison are analysed and this leads to the proposal of a number of candidate optimisations that are implemented. In the last section we give some conclusions.

4.2 The Sapl programming language

Sapl stands for **S**imple **A**pplication **P**rogramming **L**anguage. The basic version of

SapI has function application as only operation. SapI is a simple functional programming language that can be used as an intermediate formalism for the interpretation of functional programming languages like Haskell and Clean. The main difference between SapI and the intermediate formalisms normally used for these languages is the absence of algebraic data types and constructs for pattern matching in SapI. This makes SapI a compact and simple language. More details about SapI can be found in [JKP06] (chapter 3).

In chapter 3 we also showed how to represent data types and pattern-based function definitions in SapI. Here we shortly repeat the definition of the list data type together with the *length* function.

$$\begin{array}{lcl} Nil & = & \lambda f\ g\ \rightarrow\ f \\ Cons\ x\ xs & = & \lambda f\ g\ \rightarrow\ g\ x\ xs \\ length\ ys & = & ys\ 0\ (\lambda x\ xs\ \rightarrow\ 1\ +\ length\ xs) \end{array}$$

Now consider a pattern based Haskell function like *mappair*.

$$\begin{array}{lll} mappair\ f\ Nil & zs & = Nil \\ mappair\ f\ (Cons\ x\ xs) & Nil & = Nil \\ mappair\ f\ (Cons\ x\ xs) & (Cons\ y\ ys) & = Cons\ (f\ x\ y)\ (mappair\ f\ xs\ ys) \end{array}$$

This definition can be transformed to the following SapI function (using the above definitions of *Nil* and *Cons*).

$$\begin{array}{lcl} mappair\ f\ as\ zs & = & as\ Nil\ (\lambda x\ xs\ \rightarrow\ zs\ Nil\ (\lambda y\ ys\ \rightarrow \\ & & Cons\ (f\ x\ y)\ (mappair\ f\ xs\ ys))) \end{array}$$

4.3 An Interpreter for SapI

The only operations in SapI programs are function application and a number of (built-in) integer operations. Therefore, an interpreter can be kept small and elegant. The interpreter is based on straightforward graph-reduction techniques as described in Peyton Jones [PJ87], Plasmeijer and van Eekelen [PvE93] and Kluge [Klu04]. We assume that a pre-compiler has eliminated all algebraic data types and pattern definitions (as described earlier), removed all let(rec)- and where- clauses and lifted all lambda expressions to the global level. Only constant let-expressions are allowed to enable sharing and cyclic expressions. The interpreter is only capable of executing function rewriting and the basic operations on integers. The most important features of the interpreter are:

- It uses 4 types of memory Cells. A Cell corresponds to a node in the syntax tree and is either an: Integer, (Binary) Application, Variable or Function Call. To keep memory management simple, all Cells have the same size. A type byte in the Cell distinguishes between the different types. Each Cell uses 12 bytes of memory.

- The memory heap consists only of Cells. The heap has a fixed size, definable at start-up. We use mark and sweep garbage collection.
- It uses a single argument stack containing only references to Cells. The C (function) stack is used as the dump for keeping intermediate results when evaluating strict functions (numeric operations only).
- The state of the interpreter consists of the stack, the heap, the dump, an array of function definitions and a reference to the node to be evaluated next. In each state the next step to be taken depends on the type of the current node: either an application node or a function node.
- It reduces an expression to head-normal-form. The printing routine causes further reduction. This is only necessary for arguments of curried functions.

The interpreter pushes arguments on the stack until a function call is met. In that case the function body is instantiated while the arguments are substituted, the top application node is overwritten and evaluation continues with the new expression until we arrive at a curried call or an integer value.

4.3.1 Optimisations in the Interpreter

The interpreter can be optimised in several ways. Simple optimisations are the use of a more efficient memory representations of function calls with 1 or 2 arguments and the marking of curried calls (if possible) to avoid the useless evaluation of them. Applying these optimisations result in speed-ups up to 50%.

A more significant optimisation can be realized by marking the application of a function representing an algebraic data type element to its arguments by the keyword *select* (semantically equivalent to the identity function). This triggers the interpreter not to instantiate the entire function body at once, but first to evaluate the data type and only *select* and instantiate the relevant part of the remainder expression (more details can be found in [JKP06]) (chapter 3).

As a last optimisation, anonymous functions that are the argument of a *select* are not lifted to the global level, but are called inline (see chapter 3).

As an example we show how the *select* optimisation is applied in the *mappair* function (the lambda expressions in this example are not lifted to the global level).

$$
\begin{aligned}
&\mathit{mappair}\ f\ \mathit{as}\ \mathit{zs}\ = \\
&\qquad\qquad \mathit{select}\ \mathit{as}\ \mathit{Nil}\ (\lambda\, x\ \mathit{xs}\ \rightarrow \\
&\qquad\qquad \mathit{select}\ \mathit{zs}\ \mathit{Nil}\ (\lambda\, y\ \mathit{ys}\ \rightarrow\ \mathit{Cons}\ (f\ x\ y)\ (\mathit{mappair}\ f\ \mathit{xs}\ \mathit{ys})))
\end{aligned}
$$

The *select* optimisation is essential and may result in speed-ups of more than 100 times. Normally the *select* annotations are added while translating Haskell or Clean programs to SapI, but it is possible to add the *select* annotations during a compile time analysis of a SapI program. During this analysis it is determined where applications of data type functions to other arguments occur. This analysis can only be performed in case of complete programs and not for separately compiled files

(modules). For example, if we consider the definition of *mappair* in isolation it is not clear that *as* and *zs* are *selectors*. One needs an example of the usage of *mappair* to determine that.

4.3.2 Considerations

The interpreter without the *select* optimisation and the integer operations is a pure graph reductor. The only operations are graph reduction (push arguments on the stack until a function call is met) and graph instantiation (copy a function body and meanwhile substitute the arguments from the stack).

Numeric operations are strict in the sense that the arguments have to be evaluated before the operation can be performed. The same holds for the *select* optimisation. Also in this case the first argument of *select* has to be evaluated before the operation (selection of the appropriate argument) can take place. The optimisation prevents the instantiation of large graphs. In the remainder of this paper we show that many of the optimisations we implement in the compiler involve the use of strictness to prevent the instantiation of unnecessary graphs.

4.4 A Sapl Compiler

We present two versions of the compiler: a basic version and an optimised version. The optimisations are a result of an analyses of the performance of the basic version for a number of benchmarks.

The benchmarks we use for the comparison are the same we used for comparing the Sapl interpreter with several other interpreters and compilers in [JKP06] (chapter 3). We briefly repeat the description of the benchmarks (their code can be found in [SAP]):

1. **Prime Sieve** The prime number sieve program (*primes !! 5000*).
2. **Symbolic Primes** Prime sieve using Peano numbers (*sprimes !! p280*).
3. **Interpreter** A small Sapl interpreter. As an example we coded the prime number sieve for this interpreter and calculated the 100th prime number.
4. **Fibonacci** The (naive) Fibonacci function, calculating *fib 35*.
5. **Match** Nested pattern matching (5 levels deep), repeated 2000000 times.
6. **Hamming** The generation of the list of Hamming numbers (a cyclic definition) and taking the 1000th Hamming number, repeated 10000 times.
7. **Twice** A higher order function (*twice twice twice twice (add 1) 0*), repeated 400 times.
8. **Queens** Number of placements of 11 Queens on a 11 * 11 chess board.

9. **Knights** Finding all Knight tours on a 5 * 5 chess board.

10. **Parser Combinators** A parser for Prolog programs based on Parser Combinators parsing a 17000 lines Prolog program.

11. **Prolog** A small Prolog interpreter based on unification only (no arithmetic operations), calculating all descendants in a six generations family tree.

12. **Sorting** Quick Sort (20000 elements), Merge Sort (200000 elements) and Insertion Sort (10000 elements).

Three of the benchmarks (*Interpreter*, *Prolog* and *Parser Combinators*) are realistic programs, the others are typical benchmark programs that are often used for comparing implementations.

We use C++ as a target language for our compiler. We do not use the object oriented properties of C++ (classes and member functions). But we use some specific features of C++ like reference variables. In all versions of the compiler there is a one-to-one correspondence between Sapl and C(++) functions. Because we want to use the compiler for educational purposes we strive at readable and understandable generated code.

The generic structure of a translated function is:

int funcname(Reduct t) { *instantiate_body*; *return eval_body*; }

Here *funcname* is the name of the translated Sapl function. We assume that all arguments of a function are already on the stack when the function is called. The argument t of the function is a reference to the top node of the call for this function. To enable sharing we have to overwrite this top node with the result of the function. The function returns an integer. This is because functions that result in an algebraic data type have to return the selection number needed in a *select* construction. Because we want to use the same type signature for all functions, all functions have to return an integer. Note that we cannot give the C function the same arguments as the original function because we can make curried calls to a function which is, of course, not possible in C.

4.4.1 A Basic Sapl Compiler

If we take a closer look at the Sapl interpreter, the most obvious candidate for compilation is the instantiation of function bodies. The interpreter uses a recursive function *instantiate* to copy the body and substitute the arguments. It is straightforward to generate C++ code that does this instantiation directly.

Due to the *select* optimisation the body of a function containing a *select* is not copied at once but in parts. Therefore, in the translation to C++, we add the control structure (using *if* or *switch/case* statements) to enable this copying in parts. Also the generation of this control structure is entirely straightforward.

Examples

As an example consider the translation of the functions *sieve* and *el* from the prime number sieve program.

$$\begin{aligned} &sieve\ xs = cons\ (hd\ xs)\ (sieve\ (filter\ (nmz\ (hd\ xs))\ (tl\ xs))) \\ &el\ n\ xs\ \ = select\ xs\ error\ (\lambda\ a\ as\ \rightarrow\ if\ (eq\ n\ 0)\ a\ (el\ (sub\ n\ 1)\ as)) \end{aligned}$$

The translation of *sieve* results in:

```
int sieve(Reduct t) {
  testmem();
  setCell(t,SELB,newR(OPFUNC,get(0),0,9),newR(OFUNC,
     newR(BPFUNC,newR(OPFUNC,newR(OPFUNC,get(0),0,9),0,7),
     newR(OPFUNC,get(0),0,10),3),0,5),2);
  pop(1);
  return eval(t);
}
```

testmem() checks if garbage collection is necessary. This check is done before every body instantiation. *setCell(t,...)* overwrites *t*. Although the *setCell* call looks quite complicated the only thing that is happening here is the allocation of a new graph in memory. Due to the memory optimisations for applications with one and two arguments and the marking of curried applications there are a large number of cell types (*SELB*, *OPFUNC*, etc.). *get(i)* returns a reference to the *i-th* element on the stack. *pop(i)* removes *i* elements from the stack. In the last line *eval(t)* recursively starts evaluating the resulting expression. The only thing the *eval* function does is pushing arguments on the stack and calling the resulting function.

The translation of *el* results in:

```
int el(Reduct t) {
  Reduct res = get(1);
  if(eval(res)) {
    pushs(res->r); pushs(res->l);
    testmem();
    res = newR(BINOPER,get(2),newR(NUM,Reduct(0),0),5);
    if(eval(res)) {
      testmem();
      setCell(t,BPFUNC,newR(BINOPER,get(2),
                      newR(NUM,Reduct(1),0),1),get(1),4);
      pop(4);
    }
    else {overwrite(t,get(0)); pop(4);}
  }
  else    {setCell(t,SFUNC,0,Reduct(0),0);   pop(2);}
  return eval(t);
}
```

In this example we see that the control structure of the original function is clearly reflected in the C++ function. In the first line *xs* is assigned to *res*. *res* is evaluated.

Table 4.1: Comparison Speed of Basic Compiler (Time in seconds)

	Pri	Sym	Inter	Fib	Match	Ham	Twi	Qns	Kns	Parse	Plog	Qsort	Isort	Msort
SAPL Int	6.1	17.6	7.8	7.3	8.5	15.7	7.9	6.5	47.1	4.4	4.0	16.4	9.4	4.4
SAPL Bas	4.3	13.2	6.0	6.5	5.9	9.8	5.6	5.1	38.3	3.8	2.6	10.1	6.7	2.6
GHC	2.0	1.7	8.2	4.0	4,1	8.4	6.6	3.7	17.7	2.8	0.7	4.4	2.3	3.2
GHC -O	0.9	1.5	1.8	0.2	1.0	4.0	0.1	0.4	5.7	1.9	0.4	3.2	1.9	1.0
Clean	0.9	0.8	0.8	0.2	1.4	2.4	2.4	0.4	3.0	4.5	0.4	1.6	1.0	0.6

In case the result is a *cons* (returns 1) the arguments of *cons* are pushed on the stack. Next the expression *eq n 0* is instantiated and evaluated. If $n \mathrel{!}= 0$ the expression *el (sub n 1) xs* is instantiated and the stack is cleared. In case $n == 0$, t is overwritten with x. Also in this case the stack is cleared. The last *else* handles the case that the list was *nil*.

We conclude that the basic compiler results in concise code that clearly reflects how the graph-reduction process is conducted. For a function acting on a data structure with 3 or more cases a C++ *switch* statement is generated. The adaptations to the interpreter needed to generate the C++ functions are modest. An interesting aspect is that the resulting C++ functions are integrated in the interpreter environment. The only difference for the user is the increase in speed (and an extra compilation round before starting the interpreter).

Although the Basic Compiler compiles to C++, it is essentially still an interpreter. The way graphs are reduced is the same as in the original interpreter.

In the remainder of this paper we sometimes abbreviate the instantiation of graphs with: `instantiate('expression')` or `overwrite(t,'expression')`.

4.4.2 Performance of the Basic Compiler

In Table 4.1 we compare the performance of the basic compiler with that of the interpreter and of the GHC and Clean compilers. If we compare the basic compiler with the interpreter we see that the basic compiler is about 40% faster (speed-ups between 10 and 60%).

If we compare the basic compiler with GHC (without optimiser) we see that in three cases (*Interpreter*, *Mergesort* and *Twice*) the basic SapI compiler is already faster. In the other cases GHC is mostly less than 2 times faster. Relatively slow SapI benchmarks are *Symbolic Primes* (7 times) and *Prolog* (3.7 times).

Comparing the basic compiler with GHC -O and Clean we measure large differences in performance, varying from 10% faster (compared to *Parser Combinators* in Clean) to more than 30 times slower (*Fibonacci* for Clean, GHC -O and *Twice* for GHC -O).

4.4.3 Analysis of Basic Compiler

Compared with GHC (without optimiser) the Basic Compiler is already doing a reasonable job. The only poor performing benchmark is *Symbolic Primes*. This

is an atypical program, because there is no integer arithmetic in this example and the functions bodies are all very small. For Sapl this means a lot of interpretation overhead. More important, the performance dominating functions *Mod* and *Subtract* are tail recursive. In the sequel we show that, using tail recursion optimisation, the performance of this benchmark can be improved significantly.

If we take a closer look at the benchmarks for the comparison with GHC -O and Clean, we see that there is only one benchmark that performs good in this comparison: *Parser Combinators*. This is the most 'functional' of all benchmarks in the sense that it manipulates mostly higher order functions. For a compiler this means that a lot of closures must be maintained. Closures are represented by structures comparable to the graphs in Sapl. Every compiler should analyse (destruct) these closures at a certain moment in a way similar to the way the Basic Sapl compiler does this.

The worst performing benchmarks are: *Symbolic Primes*, *Fibonacci*, *Queens* and *Twice*.

- **Symbolic Primes** we already discussed above. It contains a number of tail recursive functions for which Sapl does no optimisations yet.

- **Fibonacci** is a purely numeric function (numeric arguments and numeric operations only). In Sapl every time the function is called in the recursion, a complete instantiation of the function body is made (on the heap). The Clean and GHC -O compilers optimise this function and do not use closures but instead only use the stack to execute it.

- **Queens** has a number of numeric sub-expressions and has a (hidden) tail recursion in the function *safe*. Also in this case Clean and GHC -O use strictness analysis to eliminate the building of many closures.

- **Twice** is a special case. GHC -O has a much better performance than both Sapl and Clean. If we study the generated code for GHC -O we see that some very specific inline optimisations are made. We did not make any special optimisations for this example.

Conclusions and Plan for Optimisations

The basic compiler has already a nice performance for programs manipulating mostly higher order functions. Therefore, we may expect that the poorer performance is caused by the overhead involved in building instantiations (closures) that are not really necessary. The optimisations we apply are aimed at either preventing the building of closures or at building smaller closures. In the light of the discussion above we focus on tail recursive functions and on numeric functions and (sub)expressions, also because they can be recognized and optimised easily. But before that we look at some straightforward optimisations.

4.4.4 Reducing the size of closures and removal of interpretation overhead

Consider the following function g:

$$g\ a\ b\ c\ d\ =\ f\ a\ (h\ b\ c)\ d$$

In the basic compiler this is compiled to:

```
int g(Reduct t) {
  testmem();
  setCell(t,APP,newR(APP,newR(APP,newR(FUNC,0,0,2),get(0)),
            newR(BFUNC,get(1),get(2),1)),get(3));pop(4);
  return eval(t);
}
```

In the body of g a large instantiation is built for which *eval* is called immediately. *eval* pushes the arguments of f on the stack and calls the function f. But if we already know this, we can explicitly code the pushing of the arguments and the call to f. In this way we both save instantiation and interpretation overhead.

```
int g(Reduct t) {
  testmem();
  Reduct a0,a1,a2;
  a0 = get(0);
  a1 = newR(BFUNC,get(1),get(2),1);
  a2 = get(3);
  pop(4);
  pushs(a2);pushs(a1);pushs(a0);
  return f(t);
}
```

In this example the number of allocated nodes is reduced from 4 to 1!

We apply this optimisation whenever possible. This means that an, at compile time, known function should be called with enough arguments.

4.4.5 Numerical Functions and Expressions

If a function has numeric arguments only and its body is a purely numerical expression we can avoid the creation of closures altogether. Consider for example the Fibonacci function:

$$fib\ n\ =\ if\ (n\ <\ 2)\ 1\ (fib\ (n\ -\ 1)\ +\ fib\ (n\ -\ 2))$$

The Basic SapI compiler translates this to:

```
int fib(Reduct t) {
  Reduct res;
  testmem();
  res = newR(BINOPER,newR(NUM,Reduct(2),0),get(0),7);
```

```
  if(eval(res)) {
    testmem();
    setCell(t,BINOPER,newR(OPFUNC,newR(BINOPER,get(0),
                                  newR(NUM,Reduct(1),0),1),0,35),
                      newR(OPFUNC,newR(BINOPER,get(0),
                      newR(NUM,Reduct(2),0),1),0,35),0);
    pop(1);
  }
  else {
    setCell(t,NUM,Reduct(1),0);
    pop(1);
  }
  return eval(t);
}
```

In the optimised translation *fib* is translated to:

```
int fibh(int n) {
  if (n < 2) return 1;
  else return fibh(n-1) + fibh(n-2);
}

int fib(Reduct t) {
  eval(get(0));
  setCell(t,NUM,Reduct(fibh(getNum(get(0)))),0);
  pop(1);
  return 0;
}
```

fibh is a pure C++ function without any instantiations of cells and *fib* is a wrapper function for calling *fibh* from a functional context. The speed-up obtained in this way is more than 30 times. This version of *fib* now has a performance comparable to that of Clean and GHC -O.

Numerical expressions with a Boolean result

A special case of numeric expressions are those with a Boolean result. They often occur in the condition of an *if* statement. The *el* function we studied already before is an example of such a function. Using the numeric expression optimisation the compiled function becomes:

```
int el(Reduct t) {
  Reduct res = get(1);
  if(eval(res)) {
    pushs(res->r); pushs(res->l);
    eval(get(2));
    if(getNum(get(2) == 0){overwrite(t,get(0)); pop(4);}
    else {
      testmem();
```

```
      setCell(t,BPFUNC,newR(BINOPER,get(2),
                      newR(NUM,Reduct(1),0),1),get(1),4);
      pop(4);
    }
  }
  else    {setCell(t,SFUNC,0,Reduct(0),0);   pop(2);}
  return eval(t);
}
```

This saves allocation and interpretation overhead.

4.4.6 Optimising Tail Recursion Functions

Replacing tail recursions by while loops are a common optimisation also applied for strict functional and imperative languages. In these cases the optimisation is used to eliminate calling and stack overhead. But in the lazy functional context we have an extra benefit. Also the building of a closure (and the destruction of it) for the recursive call is prevented. Therefore, the speed-up is even higher.

Simple tail recursive functions have the form:

$$f\ a\ arg = if\ (cond\ a)\ (default\ a\ arg)\ (f\ (dec\ a)\ (update\ a\ arg))$$

The recursion runs over a. For the sake of simplicity we assume that there is only one other argument. The function contains a simple *if* construction at the top level. In the *else* case the same function is called with an a argument that is in some way smaller than the original argument. We compile this function to a C++ function containing a while-loop.

```
int f(Reduct t) {
  Reduct res  = instantiate('cond a');
  Reduct &a   = get(0);
  Reduct &arg = get(1);
  while(eval(res)) {
      arg  = instantiate('update a arg');
      a    = instantiate('dec a');
      res  = instantiate('cond a');
    }
  overwrite(t,'default a arg'); pop(2);
  return eval(t);
}
```

Note that we use reference variables for a and *arg*, so they remain on the Sapl stack, which is necessary for garbage collection purposes. In the while loop we instantiate the new versions of the arguments and the condition. The while condition determines if the recursion is finished. Because the arguments of the tail recursion are maintained by variables we can easily optimise numeric or Boolean arguments (see Subsection 4.4.5). As an example, consider the function *length* (note the use of an accumulating parameter).

$$length\ n\ xs\ =\ select\ xs\ n\ (\lambda\ a\ as\ \rightarrow\ length\ (n\ +\ 1)\ as)$$

This function is translated to:

```
int length(Reduct t) {
  eval(get(0));
  int n = getNum(get(0));
  Reduct &xs = get(1);
  while(eval(xs)) {
    n = n + 1;  xs = xs -> r;
  }
  overwrite(t,newR(NUM,Reduct(n),0)); pop(2); return 0;
}
```

Here the argument n is numerical and therefore assigned to the *int* variable n. The expression *n+1* is not instantiated, but directly translated to C. This saves an instantiation and a reduction. After the while loop we have to wrap the numeric result in a cell.

Note that this function also does not build the large closure *0+1+1+1+..* that is only evaluated at the end, which happens in the SapI interpreter and the Basic Compiler. In this way a basic form of strictness analysis is realized. Furthermore, there is another optimisation. The arguments of *Cons* are not pushed onto the stack, but can be found as the left and right child of *xs*. In the while loop of this function no instantiations are made any more!

A tail recursion may also run over several arguments. In that case the condition is a conjunction of all the conditions. As an example, consider the following definitions of *Zero* and *Suc* and the tail recursive function *Sub* running over 2 arguments, all occurring in the *Symbolic Primes* benchmark:

$$Zero\ f\ g\ =\ f$$
$$Suc\ n\ f\ g\ =\ g\ n$$
$$Sub\ m\ n\ =\ select\ n\ m\ (\lambda\ pn\ \rightarrow\ select\ m\ Zero\ (\lambda\ pm\ \rightarrow\ Sub\ pm\ pn))$$

Sub is translated to:

```
int Sub(Reduct t) {
  Reduct &m = get(0);
  Reduct &n = get(1);
  while(eval(n) && eval(m)) {
    m = m -> l;
    n = n -> l;
  }
  if(eval(n)) {
    overwrite(t,'Zero');pop(2);return 0;
  }
  else {
    overwrite(t,'m');pop(2);return eval(t);
  }
}
```

Note that after the while we have 'to check' why the loop stopped to return the result of the right stopping case. Note also that we made use of the fact that the && operator in C++ is conditional (lazy). Again, no instantiations are made in the while loop.

Tail recursion that run over 3 or more variables are handled in a similar way.

Hidden Tail Recursions

Sometimes a function can be easily converted to a tail recursion. For example, in the *safe* function used in the *Queens* benchmark an *and* condition with a recursive call to *safe* itself occurs.

$$\begin{aligned} safe\ xs\ d\ x\ =& select\ xs\ True \\ & (\lambda\ y\ ys\ \rightarrow\ and\ (and\ (neq\ x\ y)\ (neq\ (add\ x\ d)\ y)) \\ & \qquad (and\ (neq\ (sub\ x\ d)\ y)\ (safe\ ys\ (add\ d\ 1)\ x))) \end{aligned}$$

safe is translated to:

```
int safe(Reduct t) {
  Reduct xs = get(0);
  eval(get(1)); eval(get(2));
  int d = getNum(get(1));
  int x = getNum(get(2));
  int y;
  while(eval(xs) && (eval(xs -> l),y = getNum(xs -> l),x != y) &&
                    (x + d != y) && (x - d != y)) {
    xs = xs -> r;
    d = d + 1;
  }
  if (eval(xs)) {
      setCell(t,FALSE,0,0);
      pop(3);
      return 1;
    }
    else {
      setCell(t,TRUE,0,0);
      pop(3);
      return 0;
  }
}
```

Also in this case we make use of the conditionality of the && operator in C++.

4.4.7 Results and Discussion

Table 4.2 gives the results of the comparison of the optimised compiler with the other compilers and the Interpreter. We see that the optimisations result in a significant speed-up in almost all cases. We briefly discuss the speed-up obtained for the benchmarks.

Table 4.2: Comparison Speed of Optimized Compiler (Time in seconds)

	Pri	Sym	Inter	Fib	Match	Ham	Twi	Qns	Kns	Parse	Plog	Qsort	Isort	Msort
SAPL Int	6.1	17.6	7.8	7.3	8.5	15.7	7.9	6.5	47.1	4.4	4.0	16.4	9.4	4.4
SAPL Bas	4.3	13.2	6.0	6.5	5.9	9.8	5.6	5.1	38.3	3.8	2.6	10.1	6.7	2.6
SAPL Opt	2.6	1.8	3.3	0.2	3.1	5.9	4.5	0.9	18.0	2.9	1.3	6.0	2.5	1.2
GHC	2.0	1.7	8.2	4.0	4,1	8.4	6.6	3.7	17.7	2.8	0.7	4.4	2.3	3.2
GHC -O	0.9	1.5	1.8	0.2	1.0	4.0	0.1	0.4	5.7	1.9	0.4	3.2	1.9	1.0
Clean	0.9	0.8	0.8	0.2	1.4	2.4	2.4	0.4	3.0	4.5	0.4	1.6	1.0	0.6

1. **Prime Sieve** Speed-up 1.65: numeric optimisations and a tail recursion in *elem*.
2. **Symbolic Primes** Speed-up 7.3: tail recursions in functions *Mod*, *Gt*, *Neq* and *Sub*.
3. **Interpreter** Speed-up 1.82: tail recursions in *length*, *drop* and *elem* and several small numeric optimisations.
4. **Fibonacci** Speed-up 33: purely numeric function.
5. **Match** Speed-up 1.9: numeric optimisations.
6. **Hamming** Speed-up 1.66: small numeric optimisations.
7. **Twice** Speed-up 1.24: small numeric optimisations.
8. **Queens** Speed-up 5.7: tail recursion in *safe* and several numeric optimisations.
9. **Knights** Speed-up 2.1: numeric optimisations.
10. **Parser Combinators** Speed-up 1.3: small numeric optimisations and minor tail recursions.
11. **Prolog** Speed-up 2.0: tail recursions in several (minor) functions and some numeric optimisations.
12. **Sorting** Quick Sort (1.7), Merge Sort (2.2) and Insertion Sort (2.7): numeric optimisations.

Even for the higher order examples *Twice* and *Parser Combinators* there is a (small) speed-up due to the numeric optimisations. The greatest speed-up is obtained for the *Fibonacci* benchmark. An interesting speed-up is obtained for the *Symbolic Primes* benchmark. This result could be obtained because the functions *Mod* and *Sub* are tail recursive and dominate the performance of the benchmark. Also for *Queens* a high speed-up is obtained because the tail recursive *safe* function dominates the performance.

Compared with GHC the optimised compiler is faster in almost all cases. Only for *Primes*, *Prolog* and *QSort* GHC is slightly faster. For *Fibonacci*, *Interpreter*,

Queens and *Mergesort* the optimised SapI compiler is much faster (more than 2.5 times).

Compared with GHC -O we see that only for *Twice* GHC -O is an order of magnitude faster (45 times). The GHC -O optimiser recognizes the repetition in this higher order function and replaces it with an iteration. Note that GHC -O is also much faster than Clean in this case. In all other cases the difference is less than 3 times and in several cases SapI is even competitive. On the average the difference in performance stays within a factor of 2.

Compared with Clean we see that the greatest difference in performance stays within a factor of 6 (*Knights*). On the average Clean is about 2.5 times faster. For *Parser Combinators* the SapI compiler is faster (1.5 times).

Considering only the more realistic applications (*Interpreter*, *Parser Combinators* and *Prolog*) we see that for *Parser Combinators* the SapI compiler has competitive performance. For *Interpreter* the SapI compiler is competitive with GHC and GHC -O but is 4 times slower than Clean. In case of *Prolog* the SapI compiler is significant slower than all others. This is not surprising, because the performance dominating function *unify* in *Prolog* cannot be optimised with the techniques used in the SapI compiler. Here more sophisticated optimisations based on strictness analyses are needed.

4.5 Conclusions

In this paper we presented a compiler for lazy functional languages for educational and experimental use, based on a straightforward interpreter. For optimising this compiler we did not use the more sophisticated techniques normally used for compilers but took a more opportunistic approach, applying only two easy to detect and apply optimisations. This has as an advantage that the generated functions have a simple structure. This makes it possible for the user to inspect how the optimisations are applied and it also enables the user to experiment with other (hand-made) optimisations.

The compiler generates comprehensible C++ code that gives the programmer clear insight in how constructs from functional programming languages are implemented. This in contrast with the GHC compiler that also uses C as an intermediate language, but for which the generated C code is difficult to understand and looks more like assembly than like an ordinary C program.

We have learned that sometimes applying simple optimisations result in significant speed-ups (e.g *Fibonacci* and *Symbolic Primes*), but in other cases the optimisations do not suffice. In these examples (e.g. *Prolog*) the difference with Clean and GHC is still too big. We also learned that optimising a function always boils down to trying to prevent the building of unnecessary graphs (closures). In our approach this was always realized by replacing 'functional code' by 'imperative code' in the generated C++ functions.

An interesting question is, if it is possible to extend the set of optimisations in such a way that the performance becomes competitive to that of GHC and Clean in all cases while maintaining readable and comprehensive generated code.

Chapter 5

Embedding a Web-Based Workflow Management System in a Functional Language

[1] ***Abstract*** Workflow management systems guide and monitor tasks performed by humans and computers. The workflow specifications are usually expressed in special purpose (graphical) formalisms. These formalisms impose severe restrictions on what can be expressed. Modern workflow management systems should handle intricate data dependencies, offer a web-based interface, and should adapt to dynamically changing situations, all based on a sound formalism. To address these challenges, we have developed the iTask system, which is a novel workflow management system. We *entirely embed* the iTask specification language in a modern general purpose *functional* language, and *generate* a complete workflow application. In this paper we report our experiences in developing the iTask system. It not only inherits state-of-the-art programming language concepts such as generic programming and a hybrid static/dynamic type system from the host language Clean, but also offers a number of novel concepts to generate complex, real-world, multi-user, web based workflow applications.

5.1 Introduction

Workflow Management Systems (WFMS) are computer applications that coordinate, generate, and monitor tasks performed by human workers and computers. Workflow *specification* plays a dominant role in WFMSs: the work that needs to be done to achieve a certain goal is specified as a structured and ordered collection of tasks that are assigned to available resources at run-time. In many WFMSs, the workflow specification only determines the framework for the workflow application, i.e. a *partial* workflow *application*. In other WFMSs one has to provide much details in the workflow specification. In both approaches substantial coding is required to complete the workflow application. In general, this results in complex distributed, multi-user and heterogeneous applications that are hard to maintain.

[1] Originally published as [JPKA10]

In this paper, we report on our experience in designing, building, and deploying the iTask system [PAK07], which is a novel WFMS based on state-of-the-art programming language concepts with firm roots in functional programming. We developed the iTask system, because of a number of perceived issues with contemporary WFMSs. Their complex nature makes it very hard to correctly create a complete application from the partial application that is generated by them. Furthermore, contemporary WFMSs use special purpose (mostly graphical) specification languages to enable the rapid development of a workflow framework. Unfortunately, these formalisms often offer limited expressiveness. First, *recursive definitions* are commonly inexpressible, and there are only limited ways to make *abstractions*. Second, workflow models usually only describe the *flow of control*. Data involved in the workflow is mostly maintained in databases and is extracted or inserted when needed. Consequently, workflow models cannot easily use this data to parametrize the flow of work. This results in more or less pre-described workflows that cannot be dynamically adapted. Third, these dedicated languages usually offer a fixed set of *workflow patterns* [AHKB02]. However, in the real world work can be arranged in many ways. If it does not fit in a (combination of) pattern(s), then the workflow specification language probably cannot cope with it either. Fourth, and related, is the fact that functionality that is not directly related to the main purpose of the special purpose language is hard to express. To overcome this limitation, one either extends the special language or interfaces with code written in other formalisms. In both cases one is better off with a well designed general purpose language.

For the above reasons, the iTask system is a *domain specific language* that is *embedded* in a textual, formal general purpose *programming language* as a workflow specification language. This allows us to address all computational concerns within the workflow specification and provides us with general recursion. We use a *functional* language, because it offers a lot of expressive power in terms of modeling domains, use of powerful types, and functional abstraction. We use the *pure* and *lazy* functional programming language Clean, which is a state-of-the-art language that offers fast compiler *and* interpreter technology, generic programming features [AP02], a hybrid static/dynamic type system [VP03], which are paramount for generating systems from models in a type-safe way. Workflows modeled in the iTask system result in complete workflow applications that run on the web distributed over server and client side [PJKA08] (chapter 6). Clean and the iTask system can be found at `http://clean.cs.ru.nl/` and `http://itask.cs.ru.nl`.

The remainder of this paper is organized as follows. We present the iTask system in Sect. 5.2 and give a case study in Sect. 5.3. We discuss our experience in Sect. 5.4 and 5.5. Related work is discussed in Sect. 5.6. We conclude in Sect. 5.7.

5.2 Overview of the iTask system

The iTask system is a scientific prototype of a WFMS. It is also a real-world application that deploys and coordinates contemporary web technology. The main reason for using web technology is that WFMSs are by nature distributed, multi-user, and

heterogeneous software systems. The iTask system is a library made in the functional programming language Clean. The specifications that serve as input to the iTask system are expressed as a domain specific language embedded in Clean. We have adopted the practice in the functional programming community to provide a library offering a set of *combinator functions* and *primitive functions* to allow for compositional, higher-order, parametrized model specifications.

In order to give an impression of the combinators that a workflow engineer can use, Fig. 5.1 shows a few of the combinator functions and types that constitute the iTask domain specific language (for reasons of presentation, the types have been slightly simplified).

```
:: Task a          // Task is an opaque, parameterized type constructor

// Sequential composition:
(>>=) infixl 1  :: (Task a) (a → Task b)   → Task b   | iTask a & iTask b
return          :: a                       → Task a   | iTask a

// Splitting-joining any number of arbitrary tasks:
anyTask         :: [Task a]                → Task  a  | iTask a
allTasks        :: [Task a]                → Task [a] | iTask a

// Task assignment to workers:
class   (@:) infix 3 w :: w (String,Task a) → Task a   | iTask a
instance @: User, String
```

Figure 5.1: A snapshot of the iTask combinator functions.

A task is an expression of the opaque (hidden), parametrized type `Task a`. Here, `a` is a type parameter that can be instantiated with any conceivable first order type. It represents the type of the value that is produced by the task. Hence, a task (expression) of type `Task a` is a task that, once it has been performed, produces a value of type `a`.

Tasks can be combined *sequentially*. The infix combinator `>>=` and `return` function are the standard *monad* combinators [PJW93]. Task `t >>= f` first performs task `t`, which eventually produces a value of type `a`. This value can be used by the *function argument* `f`, which can compute any new kind of task expression based on that information. The type demands that `f` eventually produces a value of type `b`, which is also the final result of `t >>= f`. The task `return v` only produces value `v` without any effect.

Any number of tasks `ts` $= [\mathtt{t}_1 \ldots \mathtt{t}_n]$ $(n \geq 0)$ can be performed in parallel and synchronized (also known as *splitting* and *joining* of workflow expressions): `anyTasks ts` and `allTasks ts` both perform all tasks `ts` simultaneously, but `anyTasks` terminates as soon as *one* task of `ts` terminates and yields its value, whereas `allTasks` waits for completion of *all* tasks and returns their values.

Tasks can be assigned to workers. The expression `w @: (l,t)` assigns task `t` to worker `w`. Here `l` is a descriptive label (like the subject field in an e-mail message).

The infix operator `@:` is overloaded in the identification value of the worker, which can be a value of type `User` (a predefined iTask type), or by means of the user name (`String` value).

A more detailed description of these combinators is out of scope of this paper, but in Sect. 5.3 we give a complete example of a small, yet realistic and complex workflow that uses many of the above combinators. The crucial points are that first, all combinator functions are parametrized and statically type checked with the data that flows along the tasks. Second, tasks can inspect this data and change the control flow accordingly. Third, there is no limit on the type of the data that is passed along, provided that suitable generic functions (see Sect. 5.5) are available. This is expressed by means of the type class context restrictions (`| iTask ...`). Fourth, several combinators to express iteration are included in the iTask library. However, because the iTask system is a library embedded in Clean, the workflow engineer can define new combinators and even define recursive workflows if desired.

In addition to combinators that combine task expressions in new ways, the workflow engineer also needs primitive iTask functions. Fig. 5.2 shows some.

```
// Worker interaction:
enterInformation   :: question          → Task a      | html question & iTask a
updateInformation  :: question a        → Task a      | html question & iTask a
showMessage        :: message           → Task Void   | html message
chooseTask         :: question [Task a] → Task a      | html question & iTask a

// Worker administration:
chooseUsersWithRole:: question String   → Task [User] | html question
```

Figure 5.2: A snapshot of the iTask primitive combinator functions.

The archetypical primitive iTask combinator is `enterInformation q` which, when performed, presents the current worker with a form to create a new value of type `a`. Here, `q` is a guiding prompt for the worker. Fig. 5.3 gives an example of a form for the type `Person`. `updateInformation q v` is similar, except that the value `v` acts as

```
:: Person = { firstName   :: String
            , surname     :: String
            , dateOfBirth :: HtmlDate
            , gender      :: Gender
            }
:: Gender = Male | Female

enterPerson :: Task Person
enterPerson = enterInformation "Enter Information"
```

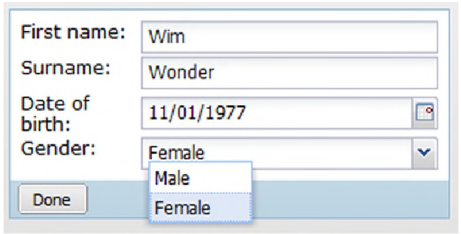


Figure 5.3: A standard form editor generated for type `Person`.

initial content of the form. The `showMessage` combinator displays a message to the user. With `chooseTask` the user can choose a task to be performed from a list of

tasks. In order to dynamically delegate work to users in the system, a workflow needs to have access to the worker administration. With the combinator function `chooseUsersWithRole` the user is given a list of current workers, and she can make a selection.

The overview of the iTask combinators here is just a selection enabling us to present the example used in Sect. 5.3. There are many more combinators that we cannot discuss here due to lack of space: combinators for the dynamic creation and control of workflow *processes*, combinators to raise and handle *exceptions* (stop a running workflow, inform all collaborators and start an alternative workflow), and combinators which allow to *change* workflows *at execution time* (replace a workflow on-the-fly by another workflow yielding a result of the same type). These features are necessary to handle realistic workflow cases.

Finally, iTask is embedded in Clean. This provides the workflow engineer with many abstraction techniques that are common practice in functional programming: tasks can be polymorphic, use higher-order functions, can be parametrized, and even higher-order workflows can be created (tasks that have tasks as parameter or result). This yields a high degree of re-usability and customization. As a final example, iTask provides a core combinator function, `parallel` that is used in the system to define many other split-join combinators such as `anyTask` and `allTasks` that were shown earlier. Its type signature is:

```
parallel :: ([a] → Bool) ([a] → b) ([a] → b) [Task a] → Task b | iTask a & iTask b
```

`parallel c f g ts` performs all tasks within `ts` simultaneously and collects their results. However, as soon as the predicate `c` holds for any current collection of results, then the evaluation of `parallel` is terminated, and the result is determined by applying `f` to the current list of results. If this never occurs, but all tasks within `ts` have terminated, then `parallel` terminates also, and its result is determined by applying `g` to the list of results.

5.3 Ordering example

To demonstrate the expressive power of iTask, we present an *ordering* example. The code presented below is a complete, executable, iTask workflow. The workflow has a recursive structure and monitors intermediate results in a parallel and-task. This case study is hard to express in traditional workflow systems. The overall structure contains the following steps (see `getSupplies` below): first, an inventory is made to determine the required amount of goods (`getAmount`) (e.g. vaccines for a new influenza virus); second, suppliers are asked in parallel how much they can supply (`inviteOffers`); third, as soon as sufficient goods can be ordered, these orders are booked at the respective suppliers (`placeOrders`).

```
getSupplies :: Task [Void]                                          1.
getSupplies = getAmount >>= inviteOffers >>= placeOrders            2.
```

Determining the required amount of goods proceeds in a number of steps:

```
getAmount :: Task Amount                                            3.
```

```
getAmount                                                                    4.
= chooseTask "Decide how much we need"                                       5.
    ["Decide yourself"    @>> enterInformation "Enter the required amount"   6.
    ,"Let others decide" @>> determineOthers]                                7.

determineOthers :: Task Amount                                               8.
determineOthers                                                              9.
=                 chooseUsersWithRole "Select institutes:" "Institute"       10.
  >>= λusers →  allTasks [  user @: ("Amount request", getAmount)            11.
                         \\ user ←- users                                    12.
                         ]                                                   13.
  >>= λothers → updateInformation "Enter required amount" (sum others)       14.
```

First, with `chooseTask` the user can choose to enter the amount herself or to ask others to determine this amount. `@>>` is used to give a task a (displayable) label. In `determineOthers`, with the task `chooseUsersWithRole` (line 10) a set of users (of type `User`) which fulfil a certain role, in this case institutes, is selected by the user. Each of the selected institutes on their turn may enquire other institutes *recursively* in parallel (using the `allTasks` combinator) how many goods they need (lines 11-13). The recursive call `getAmount` has as effect that each of the chosen institutes can ask other institutes for the same thing, and so on. Given the amount determined by others, an institute may alter the final amount it wants to have (line 14). `Amount` is a non-negative `Int`:

```
:: Amount :== Int                                                            15.
```

Once the amount of goods is established, the workflow can continue by inviting offers from a collection of candidate suppliers:

```
inviteOffers :: Amount → Task [(Supplier,Amount)]                            16.
inviteOffers needed                                                          17.
=               chooseUsersWithRole "Select suppliers:" "Supplier"           18.
  >>= λsups →  parallel enough (maximum needed) id                           19.
               [sup @:("Order request", updateInformation prompt needed      20.
                                        >>= λa → return (sup,a))             21.
               \\ sup ←- sups                                                22.
               ]                                                             23.
where enough as = sum (map snd as) ≥ needed                                  24.
      prompt    = "Request for delivery, how much can you deliver?"          25.
```

This collection is determined first (line 18). Each supplier can provide an amount (line 20). This is again done in parallel (line 19-23). The termination criterion is the `enough` predicate which is satisfied as soon as the sum of provided offers exceeds the requested amount (line 24). The function `maximum` is discussed below. Hence, the result of this task is a list of offers. Each offer is a pair of a supplier and the amount of goods that it offers to deliver. A supplier is just a user:

```
:: Supplier :== User                                                         26.
```

The total number of offered goods can differ from the required number of goods. The function `maximum` makes sure that not too many goods are ordered.

```
maximum :: Amount [(Supplier,Amount)] → [(Supplier,Amount)]                    27.
maximum needed offers   = [(sup,exact) : rest]                                 28.
where                                                                          29.
    [(sup,_) : rest]    = sortBy (λ(_,a1) (_,a2) → a1 > a2) offers             30.
    exact               = needed - sum (map snd rest)                          31.
```

With the correct list of offerings, we can place an order for each supplier. This can be expressed directly with `allTasks`:

```
placeOrders :: [(Supplier,Amount)] → Task [Void]                               32.
placeOrders offers                                                             33.
=   allTasks [sup @: ("Order placement", showMessage ("Please deliver " <+ a))  34.
             \\ (sup,a) ← offers                                               35.
             ]                                                                 36.
```

The overloaded infix operator `<+` converts its right-hand argument to a string and glues it to the given left-hand argument. It is part of the iTask system.

In order to complete the case study, the `getSupplies` workflow needs to be passed to the iTask run-time system as a workflow that returns `Void`:

```
Start :: *World → *World                                                       37.
Start world  = startEngine [workflow] world                                    38.
where                                                                          39.
    workflow = { name     = "Ordering example"                                 40.
               , label    = "Collect ordering info and make the order"         41.
               , roles    = []                                                 42.
               , mainTask = getSupplies >>= λ_ → return Void                   43.
               }                                                               44.
```

5.4 Experience with the iTask language

iTask is a prototype language. We have investigated its expressiveness by means of constructing examples as well as larger case studies, for instance a conference management system [PAK+08b]. The next step is to investigate its use in demanding environments that concern crisis-management situations, in a project with the Netherlands Defense Academy. In this section we report on our experience in using the iTask specification language.

5.4.1 iTask is built on a single, powerful, concept

In iTask, everything is constructed as (a combination of) a task. The notion of a task and the combinators we use have a clear semantics [KPA09]. A task represents work that needs to be performed, and abstracts over the way the task is composed out of sub-tasks and the order in which these sub-tasks are being evaluated. No matter how complex a task may be, for the programmer a task remains a unit of work returning a value of type (`Task a`) once the task as a whole is terminated. The result of a task can be used as input for other tasks. The coordination of tasks is defined by means of combinators.

A task represents work that needs to be performed. This work can be anything that is required by the workflow case, such as connecting to a legacy information system, calling a web service, or arbitrary foreign code. For instance, for access to information stored in standard information systems, we have developed a systematic conversion between an information model defined in e.g. ORM (Object Role Model) and Clean data type definitions. This enables the automatic conversion between values of these types and the corresponding values stored in a relational database [LP09a], without the need for explicit SQL programming. As another example, for

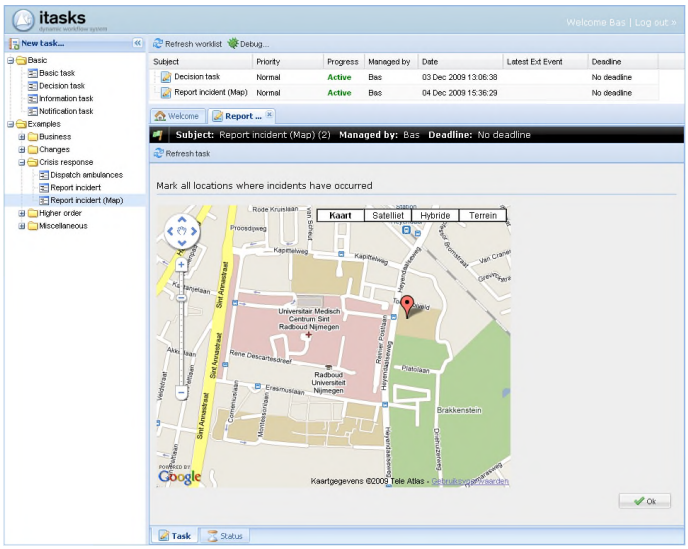


Figure 5.4: An iTask for manipulating a map

the type `GoogleMap`, the basic task `enterInformation` will show a standard Google Map in which the end user can scroll and place markers (Fig. 5.4). User manipulations of the map are automatically kept track of and are reflected in the `GoogleMap` data structure. No extra effort is needed in the workflow specification other than using the type.

In this way, everything can be considered to be a task. An iTask specification uses combinators to coordinate tasks, and hence one can use the iTask language as a web coordination language as well.

5.4.2 iTask is a declarative language

We want the specification of a workflow to be declarative and hence to abstract from details as much as possible. Given an iTask workflow specification, the iTask system automatically generates all required web forms, handles all user data entry, storage of intermediate results, task distribution to specified workers, and handles all coordination. Also the precise way information is displayed in the browser is not specified in the workflow, but delegated to the client. To further enable abstraction over lay-out, we offer several primitives in the iTask library for basic interaction steps. For instance, in addition to `enterInformation`, there are basic primitives like

`enterChoice` and `enterMultipleChoice`. The advantage of having different primitives for such basic interaction steps is that the workflow specification becomes more readable while the representation and lay-out can again be delegated to the client. Due to abstraction, the workflow engineer can concentrate on specifying the workflow. This promotes rapid prototyping of workflow applications.

5.4.3 iTask is more than Clean

iTask is an embedded domain specific language and inherits all language aspects of its host Clean. In particular, these are the strong type system, higher-order functions, lazy and strict evaluation, and the module system. All computational and algorithmic concerns can be dealt with in the Clean language. iTask is also more than Clean because workflows are inherently sequential, distributed, multi-user, concurrent systems and the Clean standard supports none of those characteristics. Also, to model realistic workflow cases, one needs to address exceptions and dynamic change. Again, these concepts are absent in native Clean (see also Sec. 5.5). Each of the required concepts of the embedded language are challenging to add to native Clean. Nevertheless, this experiment shows that it is possible to embed a workflow language in a host that offers entirely different concepts.

5.4.4 iTask has higher-order tasks

A task in Clean of type `Task a | iTask a` effectively works for all first order types `a`. In particular, it works for the type `Task` itself, which means that tasks can be higher order: the result of a task might be a task which can be dynamically and interactively constructed. In this way meta programming (doing tasks that have as goal to define new tasks) can be accomplished. A task thus created can be given as argument to other tasks which can decide to evaluate it or to use it in the construction of an even more complex task. It is very unlikely that an ad-hoc domain specific workflow language has the ability to deal with advanced notions such as higher functions and tasks, and this feature is therefore missing in all commercial workflow systems. Embedding a workflow language in a language like Clean really pays off here.

5.5 Experience with Clean as host language

In this section we focus on our experience with using Clean as host language and implementation vehicle to embed iTask. An iTask specification results in a web application. The architecture of this web application is given in Fig. 5.5.

5.5.1 Smart combinators

iTask is a workflow language and is hence inherently sequential, distributed, multi-user, and concurrent. It needs to handle exceptional situations and dynamically changing workflows. The host language Clean offers no native support for these concepts. When developing such a language in the traditional way, one would develop

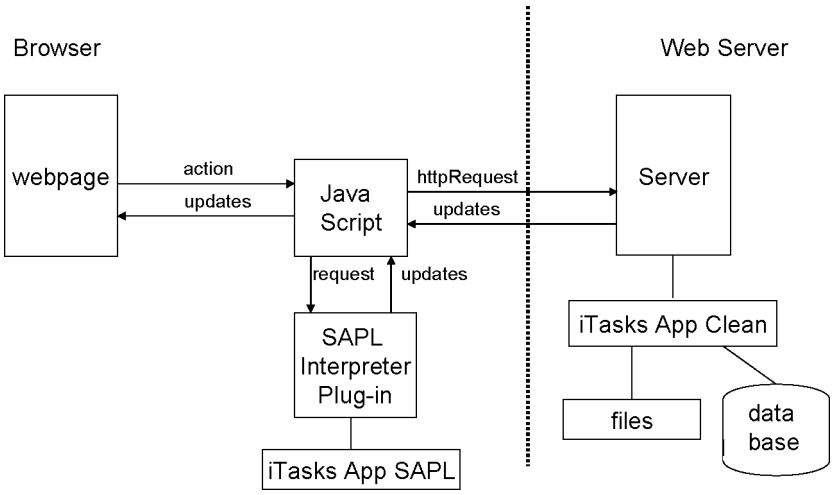


Figure 5.5: The architecture of an iTask application

a grammar, semantic rules, perhaps a type system, a compiler and/or interpreter, code generator, and so on. This is a huge amount of work. In this project we have taken a different route: when designing a language, one needs to define the semantic rules. Semantic rules can be represented in a natural way by means of functions. If one takes care in designing these rules in a compositional way, then these form a set of *smart combinator* functions. In this way one can obtain a compositional language implementation almost for free. This decreases the implementation effort of a new language significantly.

The combinators have several obligations in the iTask system. First, the combinators yield the current status (and hence GUI) at any moment during execution. For example, the iTask system can evaluate the expression `t >>= f` even if task `t` is not finished yet. The iTask system does this by creating a default value of the proper type for the whole expression `t >>= f`. In this way the status of all tasks defined in a workflow can be inspected, but only the values of the finished tasks are taken into account. Second, a new workflow is calculated by the combinators given the finished tasks. Third, each combinator stores its current state in memory and uses it for handling the next event from the participating workers.

5.5.2 Smart tasks

The iTask language is a declarative language. This implies that we want to generate as much boilerplate code as can be possibly done from an iTask specification. In iTask this has been realized by using the generic programming features of Clean [AP02]. Tasks require the availability of a collection of generic (kind indexed, type driven) functions. These generic functions are used to generate all kinds of functionality automatically, such as the generation of web forms, the handling of user updates of such forms, the storage and retrieval of information, the serialization and de-serialization of data and functions. The generic functions are predefined in the iTask

library. To use them for a certain type, however, one needs instances for that type for all the generic functions being used. As a result a task can be applied to values of *any* type, as long as instances for this type have been defined for all generic functions the task is depending on. The Clean compiler is able to generate instances for these generic functions for (almost) any (non opaque) type fully automatically. Clean is special in this respect. In Haskell e.g. generic functions can be constructed using special pre-processors like template Haskell [SP02].

It should be noted that a great deal of the facilities for which we have used generics in our project can be done in a programming language that offers introspection and code generation facilities. One significant advantage of using generics is its firm integration with the static type system of Clean.

5.5.3 Smart serialization

An iTask application is a web application that runs on the server side. This application must handle every possible user request from any possible web browser that connects with the application. After an event is handled, the web application terminates and is started all over again by the web server when new user events arrive. Hence, an iTask application needs to fully recover its previous state to compute the proper response. Conceptually, this amounts to reconstructing the *task tree* that reflects the current state of computation of the workflow. The *nodes* of a task tree are formed by the combinators in the task that is being computed, and the *leaves* of a task tree are the primitive tasks. Evaluation of a workflow boils down to *rewriting* this task tree as dictated by the combinators. The task tree can become very large. Hence, a naive implementation of task-tree rewriting for iTask applications is not realistic. Instead, we have incorporated a number of optimizations to obtain an efficient and scalable implementation. We briefly discuss two of the most important optimizations.

The first optimization is based on the observation that most rewrites affect only a local part of the task tree. Hence, for these rewrites it is not necessary to reconstruct the entire task tree, but only the subtask tree that is affected. Because an iTask application terminates after handling an event, we need to be able to store and read any subtree that is currently being rewritten. Tasks and combinators are implemented as state transition functions, hence we need to be able to store functions. Clean offers a hybrid type system, and statically typed expressions can be turned into a dynamically typed expression (of static type `Dynamic`) and the other way around. Dynamics can be stored to disk and it is even possible to read in a dynamic stored by some other Clean application.

The second optimization is based on the observation that many computations do not *have to be done* at the server side, but can also be done on the web client side. Hence, clients need to be able to run tasks, which amounts to running Clean code. To implement this, the Clean compiler generates *two* executable instances from a single source. The first instance is a Clean executable that runs on the server, and the second instance is a Sapl program to be executed by the Sapl interpreter [JKP06] (chapter 3) that is running as a Java applet at the client side. At run-time it can be

decided where to execute what. Any function or task can be shifted from server to client. For this purpose we again use dynamics in Clean to serialize functions and expressions as SapI programs at the server side and interpret them at the client side. For details we refer to [PJKA08] (chapter 6).

5.6 Related work

The WebWorkFlow project [HVV08] shares our point of view that a workflow specification is regarded as a web application. WebWorkFlow is an object oriented workflow modeling language. *Objects* accumulate the progress made in a workflow. *Procedures* define the actual workflow. Their specification is broken down into *clauses* that individually control *who* can perform *when*, what the *view* is, what should be *done* when the workflow procedure is applied, and what further workflow procedures should be *processed* afterwards. Like in iTask, one can derive a GUI from a workflow object. The main difference is that iTask is embedded in a functional language, but this has significant consequences: iTask supports higher-order functions in both the data models and the workflow specifications; arbitrary recursive workflows can be defined; reasoning about the evaluation of an iTask program is reasoning about the combinators instead of the collection of clauses.

Brambilla *et al*[BCC07] enrich a domain model (specified as UML entities) with a workflow model (specified as BPMN) by modeling the workflow activities as additional UML entities and use OCL to capture the constraints imposed by the workflow. The similarity with iTask is to model the problem domain separately. However, in iTask a workflow is a function that can manipulate the model values in a natural way, which enables us to express functional properties seamlessly (Sect. 5.3). This connection is ignored in [BCC07] and can only be done ad-hoc.

Pešić and van der Aalst [PA06a] base an entire formalism, ConDec, on linear temporal logic (LTL) constraints. Frequently occurring constraint patterns are represented graphically. This approach has resulted in the DECLARE tool [Pes08]. In iTask a workflow can use the rich facilities of the host language for computations and data declarations – such facilities are currently absent in DECLARE.

Andersson *et al*[ABE05] distinguish high level *business models* (value transfers between *agents*), low level *process models* (workflows in BPMN), and medium level *activity dependency models* (activities for value transfers of business models). Activities are *value transfer*, *assigning* an agent to a value transfer, value *production*, and *coordination* of mutual value transfers and activities. Activities are modelled as nodes in a directed graph. The edges relate activities in a way similar to [BCC07] and [PA06a]: they capture the workflow, but now at a conceptual level. A *conformance relation* is specified between a process model and an activity dependency model. Currently, there is no tool support for their approach. The activity dependency models provide a declarative foundation to bridge the gap between business models and process models. One of the goals of the iTask project is to provide a formalism that has sufficient abstraction to accommodate both business models and process models.

Vanderfeesten *et al*[VRA08] have been inspired by the *Bill-of-Material* concept from manufacturing, recasted as *Product Data Model* (PDM). A PDM is a directed graph. Nodes are product data items, and arcs connect at least one node to one target node, using a functional style computation to determine the value of the target. A tool can inspect which product data items are available, and hence, which arcs can be computed to produce next candidate nodes. This allows for flexible scheduling of tasks. Similarities with the iTask approach are the focus on tasks that yield a data item and the functional connection from source nodes to target node. We expect that we can handle PDM in a similar way in iTask. iTask adds to such an approach strong typing of product data items (and hence type correct assembly) as well as the functions to connect them.

5.7 Conclusions

In this paper we report on our experience in using the lazy, pure, functional language Clean as embedding language to specify and create web-based workflow iTask applications. Although the iTask combinator language is embedded as a library in Clean, it is by no means a *shallow* embedding, i.e. the meaning of the embedded language is not a straightforward extension of the host language. The result *is* a new language for defining workflow applications. This new language provides the workflow engineer with concepts to seamlessly merge data flow with control flow (exemplified by the >>= combinator), use higher-order tasks (tasks that can create, manipulate, and pass around tasks), in a compositional way. The evaluation order of the workflow is controlled by the iTask combinators and dictated by the needs of the workflow engineer (by using sequential and generalized parallel split-join patterns as well as recursion). It is important to observe that this evaluation order is very different from the lazy evaluation order of the host language *and* that one can add new combinators within iTask to capture other evaluation orders when needed. The iTask system is very general and serves as a coordination language to control and unify all tools that are used to realize the system. Specifications inherit the terseness of their host language.

We have used many state-of-the-art programming language techniques to obtain this result: *generic programming* to handle boilerplate code generation (including foreign code) in a type-directed way, *dynamic types* to handle arbitrary (higher-order) data structures which origin need not be the source program itself, and *higher-order functions* which permeate through the entire design, implementation, and resulting language. The entire system is statically typed. Although the boilerplate code generation aspects can be realized in other programming languages that support some form of inspection, we have shown in this project that the task of embedding a language (however alien) is one that fits functional programming languages like a glove.

Chapter 6

Declarative Ajax and Client-Side Evaluation of Workflows using iTasks

[1] ***Abstract*** Workflow systems coordinate tasks of humans and computers. The iTask system is a recently developed tool-kit with which workflows can be defined declaratively on a very high level of abstraction. It offers functionality which cannot be found in commercial workflow systems: workflows are constructed dynamically depending on the outcome of earlier work, workflows are strongly typed, and they can be of higher order. From the specification, a web-based multi-user workflow system is generated. Up until now we could only generate thin clients. All information produced by a worker triggers a round trip to the server. For real world workflows this is unsatisfactory. Modern Ajax web technology to update part of a web page is required, as well as the ability to execute tasks on clients. The architecture of any system that supports such features is complex: it manages distributed computing on clients and server which generally involves the collaboration of applications written in different programming languages. The contribution of this paper is that we integrate partial updates of web pages and client-side task evaluation within the iTask system, while retaining its approach of a single language and declarative nature. The workflow designer uses light-weight annotations to control the run-time behavior of work. The iTask implementation takes care of all the hard work under the hood. Arbitrary tasks (functional programs) can be evaluated at web clients. When such a task cannot be evaluated on the client for some reason, the system switches to server-side evaluation. All communication and synchronization issues are handled by the extended iTask system.

6.1 Introduction

A workflow system is a computer system that coordinates the work that has to be done by *human workers* in collaboration with *computers*. Workflow systems are challenging real-world applications because they need to handle many things. First of all, a workflow system has to provide a way to specify workflows: *what*

[1] Originally published as [PJKA08]

are the *tasks* that have to be done, how do these tasks *depend* on each other, and *who* should do them? The specification is used by the system at run-time for the real-time coordination and monitoring of the actual work being performed. Hence, somehow a mapping has to be made between the workflow specification and the real work that has to be done given the concrete human and software resources which are available. Daily work can be structured in quite a complex way which has direct consequences for the way tasks are depending on each other. The result of the work of one worker might determine the work of many others in both a positive or negative way. One needs a good understanding of how tasks depend on each other, and one also needs a sufficiently powerful specification formalism to express such complicated dependencies. In addition, one has to control a process which is quite dynamic: the amount and kind of work, the time it takes to do a job (ranging from split seconds to months), the number of available workers, the allocation of resources (both human, software, and hardware), they may all vary over time and may depend on the concrete work that takes place. Last but not least, one generally has to deal with a technically complicated distributed, heterogeneous environment: people working together all over the world using their own personal computer, pda's, mobile phone, and so on.

How to express this all? How to control this given a specification? It should be clear that a software system that can deal with all the above is bound to be complex. There exist many, mainly commercial, workflow systems. Examples are COSA Workflow [Sol], Business Process Manager FLOWer [PA], i-Flow 6.0 [Fuj], Staffware [TIB], Websphere MQ Workflow [IBM], and YAWL [YAW] (see also [RHAM06, Pat]). Although these systems all have their own way of dealing with the challenges mentioned above, they also have a lot in common. Usually the systems are based on Petri-nets. The advantage is that dependencies between tasks can be depicted which makes them attractive to non-experts, while these drawings can straightforwardly be mapped to a corresponding Petri-net. This Petri-net is used at run-time as scheme to control the real work to do. Furthermore the net can be used as a formal model at compile-time to determine desired properties of the specified workflow: one can calculate reachability of a certain task or determine the absence of deadlock. The kind of task dependencies one can specify in these systems, the so-called *workflow patterns*, are summarized in [AHKB02], together with a discussion of the systems mentioned above. However, the use of Petri-nets as semantic model also has big disadvantages. The nets are static and only first order: tasks cannot deliver new tasks. Hence they cannot be used to describe the dynamic way of working that takes place in the real world.

The main research question that we address in this paper has been asked to us by industry being confronted with the limitations of the current systems: can declarative programming, and functional programming in particular, provide new concepts and implementation methods and tools for workflow systems that can deal with the dynamic behavior of daily work? In answering this question, we have developed the iTask toolkit [PAK07] as a first step towards a realistic workflow system. This toolkit is a web-based combinator library written in the lazy, purely functional programming language Clean. The novel and declarative contributions

that this toolkit provides, which cannot be found in the existing commercial systems, are:

- workflows are constructed fully dynamically instead of statically: they can depend on the intermediate inputs and outputs that are yielded by workers and computations;
- workflows can be higher-order, i.e. yield partially evaluated tasks which can be passed around for further evaluation to other workers at other locations;
- workflow cases are specified as pure, strongly-typed functional expressions, using the predefined iTask combinators;
- the workflow application can handle multiple workers, multiple tasks, and multiple clients dynamically, yet everything is controlled by one, *single* Clean application running on the server;
- the specification of the workflow is *executable*; all implementation details like web-page generation, web-page handling, client-server communication and database storage handling is handled fully automatically by making intensive use of generic programming techniques [Hin00, AP02]: from the *types* being used the required *code* is *generated* fully automatically.

Tasks have to be offered to the workers in such a way that it is clear what they have to do. The iTask application generates, given the workflow specification and the work that has been done so far, an appropriate web page for each user. The key advantage of using browsers to display the work to do, is of course that they are available on any thinkable platform. No special software needs to be installed to connect iTask workflow users. Workflow systems are distributed software systems, hence it makes sense to not only deploy web technology for rendering purposes, but also for the distribution, communication, and control of tasks. However, although the web seems to be very suited for all this, it is actually technically quite difficult to realize the rendering and communication automatically from a given declarative workflow specification. Workflow systems exhibit *state*, support *multiple users*, and guide the *flow of work*. None of these concepts are readily supported by the web and hence additional software is needed for the realization. Commonly, a web application which guides a user through several working steps does not consist of one, single application. The implementation often consists of a collection of software applications and scripts, written in several languages, which somehow together do the job: one can think of HTML-code, php-scripts, Ajax-scripts, SQL-queries. Since they are commonly not generated from one single source code, it is very hard to design, implement and maintain systems which such an architecture. In the iTask system all software *is* generated from one single source in Clean. To understand what the application is doing, one only needs to look at the iTask specification. It is a specification on a very high level of abstraction which can be read as if we are dealing with an ordinary simple desktop application. We take full advantage of the fact that we are working with a pure functional language. First of all we solve the lack-of-state

problem of the web, by using generic programming techniques to store the state of the interactive elements, the iTask, *only*. Because we have the most recent states of the iTask at our disposal, we only need to rerun the function that represents the program and provide it with the most recent input action of any worker to advance to the next state. This reduces the programming burden on the workflow developer. It allows her to focus on the workflow case, rather than its implementation. Another advantage of such an approach is that one obtains a clean separation between the workflow *specification* and its *implementation*.

In this paper we explain the use and implementation of two new important features added to the iTask toolkit. In the old system, any event received from an iTask user is handled by the single iTask application on the server. It computes the next state and calculates a whole new web page for a particular user showing her the new tasks to do. New web technology such as Ajax [Gar05], makes it possible to update only a part of a page. Updating only the relevant part of a page improves the behavior of the web application in a way that resembles desktop behavior. The first feature is that we incorporate partial page updates. This is a challenge since the iTask system has to calculate dynamically which part of the page has to be updated, and this depends on the state of the task being performed and the state of the work of all other users. In most existing systems the part of the page to be updated is fixed rather than computed dynamically. Furthermore we require that the program which executes the tasks can run partially on the client instead of on the server. Client-side evaluation is essential in eliminating delays associated with the communication between server and client. The impact of this feature cannot be overestimated because it is fundamental to create coarse grained computational tasks on clients with rich interaction and quick response times (think of modern day web applications like Google Docs and gmail). There are three ways to obtain client-side evaluation of tasks in a browser: plug-ins, JavaScript, or Java code. The disadvantages of plug-ins is the explicit installation that they require. In the current iTask system we prefer Java over JavaScript since it seems better suited for the large applications that have to be run on the client.

Instead of a single server, one can also think of using several servers, as well as tasks that are migrating over the internet. Distribution of client tasks is also required when one wants to work with *distributed document repositories* that can be accessed by client workflows. This is not addressed in this paper, but will be subject of future research. The feature of client-side workflows also challenges the underlying architecture of the iTask tool-kit.

In this paper we show how these two major web techniques can nevertheless be incorporated smoothly in the iTask toolkit, while fully retaining its declarative nature:

- We rearrange the iTask toolkit in such a way that worker-tasks automatically use the asynchronous, partial page update technology that is offered by Ajax. Besides this default arrangement, we allow the workflow designer to *annotate* workflow expressions in a light-weight way to give fine-grained control of other parts of the workflow application.

- We rearrange the iTask toolkit in such a way that *any workflow task expression* can be evaluated at the client side. To indicate which parts are to be executed at the client side is only a matter of adding a simple annotation to an existing specification.

The workflow engineer can use the new contributions of the iTask tool-kit by annotating ordinary iTask applications. The implementation of these new annotations is however challenging. We wish to evaluate complete task expressions on the client instead of on the server, which requires that we can evaluate full Clean code on the client side within the browser. Worker actions can have non-local effects, and hence we need to implement some sort of *synchronization*. We show that this can be implemented without loss of the semantic model of the system, without client side evaluation and partial page updates.

The declarative nature of the iTask toolkit is retained by implementing an *evaluation strategy* that can automatically switch between client-side evaluation and server-side task rewriting if necessary. The details are presented in Sections 6.5 and 6.6; roughly speaking the system can perform tasks on the client side (within a browser) as well as on the server side (within the server application). Moreover, if client-side evaluation is no longer possible (because of a non-local effect of a remote worker, or because the local computation requires a server resource), the system automatically can continue to perform the computation on the server side. The workflow designer does not have to specify this, unlike other approaches as for instance in Hop [SGL06, LS07] (see also Sect. 6.7). This implies that the approach as described in this paper is not only more declarative, but also more robust: it can handle situations dynamically that would otherwise be considered programming errors.

The iTask toolkit has been created in Clean. A concise overview of the syntactic differences with Haskell is [Ach07]. We assume the reader is familiar with the concept of generic programming.

We start with a short overview of the iTask combinator system in Sect. 6.2. The new annotations are introduced in Sect. 6.3. Their ease of use contrasts strongly with their implementation. To understand why, we present the basic architecture of the standard implementation in Sect. 6.4. The high level specification of workflows offered by the iTask system is achieved due to the fact that the system is able to reconstruct the state of evaluation of all tasks of all users, the so-called *Task Tree*. To avoid the *Task Tree* from growing infinitely, a task expression is rewritten by its result in a similar way as function applications are rewritten by their result. This is called *Global Task-Tree Rewriting*. In Sect. 6.5 we discuss the implementation consequences of asynchronous partial page updates and introduce *Local Task-Tree Rewriting*. In Sect. 6.6 we do the same for client-side evaluation and introduce *Client-Side Local Task-Tree Rewriting*. Related work is presented in Sect. 6.7 and we conclude in Sect. 6.8.

6.2 Introduction to iTasks

In this section we give a concise overview of the iTask system. First we select the combinators that are used in this paper (Sect. 6.2.1). We present the complete code of a small, but representative case study (Sect. 6.2.2). Finally, we discuss opportunities for optimization (Sect. 6.2.3).

6.2.1 The iTask Combinators

Although the iTask system supports all common workflow patterns found in commercial workflow systems ([AHKB02] gives an excellent overview), it is beyond the scope of this paper to discuss them all. The selection of iTask combinators that we use in this paper are shown in Fig. 6.1.

```
:: Task        a
:: Pred        a :== a → (Bool, [BodyTag])
:: LabeledTask a :== (String,Task a)
:: UserId        :== Int

editTaskPred   :: a (Pred a)                  → Task a      | iData a
editTask       :: a                           → Task a      | iData a
(⋙) infix  1   :: (Task a)     (a → Task b)   → Task b      | iData b
return_V       :: a                           → Task a      | iData a
buttonTask     :: String       (Task a)       → Task a      | iData a
chooseTask     :: HtmlCode [LabeledTask a]    → Task a      | iData a
(-||-) infixr 3 :: (Task a)         (Task a)  → Task a      | iData a
(-&&-)infixr 4 :: (Task a)         (Task b)   → Task (a,b) | iData a
                                                           & iData b
(?>>) infixr 5 :: HtmlCode         (Task a)   → Task a      | iData a
(@:)  infix  3 :: UserId    (LabeledTask a)   → Task a      | iData a
```

Figure 6.1: The selection of iTask toolkit combinators

In the iTask toolkit tasks are represented by the opaque type (`Task a`). The primitive task (`editTaskPred` *a p*) generates a web form for values that have the type of the initial value *a*. The predicate *p* is used to impose further constraints on entered values (they at least have to be of correct type). Only when the worker has entered a value of correct type that also meets the given predicate the task can be finished by the worker and that value is returned. If the predicate *p* is not needed one can use `editTask` *a*. The type class restriction `| iData a` at the end of the type signature guarantees that this function works for *any* type `a` provided that all generic instances for this type of the generic functions being used are available. The compiler can automatically *derive* these instances on request of the programmer (see Sect. 6.2.2). An edit task for a string is specified as:

```
et :: Task String
et = editTask "Finished" "edit string here"
```

The initial string is `"edit string here"`. The task is finished when the user presses the button labeled `Finished`.

The iTask library uses the monadic combinators =>> and `return_V` for their standard purposes. The task `return_V "Approved"` is a task that returns the string `"Approved"` without any user interaction.

A `buttonTask` s t activates the task t after the user has pressed the button labeled by the string s. As an example: the task `yt` (`nt`) yields the string `"Approved"` (`"Rejected"`) when the user presses the button labeled `Yes` (`No`).

```
yt :: Task String
yt = buttonTask "Yes" (return_V "Approved")

nt :: Task String
nt = buttonTask "No"  (return_V "Rejected")
```

`chooseTask` *html* $[(l_0, t_0) \ldots (l_n, t_n)]$ allows the worker to pre-select one labeled task t_i from the list. After the choice, the other tasks have disappeared. For example

```
ct = chooseTask [Txt "choose"]
                [("Yes", return_V "Approved")
                ,("No",  return_V "Rejected")
                ,("Edit",et)]
```

prompts the user with the text `choose` and offers three buttons labelled `Yes`, `No`, and `Edit`. After using one of the first two buttons `ct` will be finished and deliver the indicated string. If the user presses the `Edit` button the iTask system offers the user the edit task `et`.

The expression t `-||-` u offers tasks t and u simultaneously. As soon as either one is finished first, t `-||-` u is also finished. Any work in the other task is discarded. The `-||-` combinator is very useful to express work that can be aborted by other workers or external circumstances. In

```
ot = yt -||- nt -||- et
```

the iTask system offers the task `yt`, `nt`, and `et` simultaneously. Any edit work in `et` is discarded when the user presses one of the buttons labelled `Yes` or `No`. Discarding of work is prevented in `ct` where the user chooses the task to be done before she starts editing.

Tasks can be composed sequentially by the monadic =>> operator. For instance the string resulting from `ot` can be edited until the `Done` button is pressed by executing `ot =>> editTask "Done"`.

If one really needs both results of tasks t and u, then this is expressed by t `-&&-` u, which runs both tasks to completion and returns both results. For instance, if we need a string and an integer (with default value 5) we can use the task:

```
at :: Task (String, Int)
at = ot -&&- editTask "Done" 5
```

It is useful to provide the worker with additional information *info* while she is working on a task t. This is expressed with *info* `?>>` t. Finally, any task t labeled

with l can be assigned to some user with user identification value i with i@:(l,t). We illustrate this in the next case study.

6.2.2 Case Study

The case study is a tiny *personnel administration* workflow (see also Fig. 6.2) in which two co-workers A and B simultaneously administer personnel information. In both cases they can enter information, double check their input, and submit the information. If worker A is the first to finish, then the whole workflow case terminates. When worker B finishes first, the result is given to worker A, who can inspect and adjust this value. Note that at this stage, worker A now has the choice of either finishing her own version, or decide to continue work on B's result.

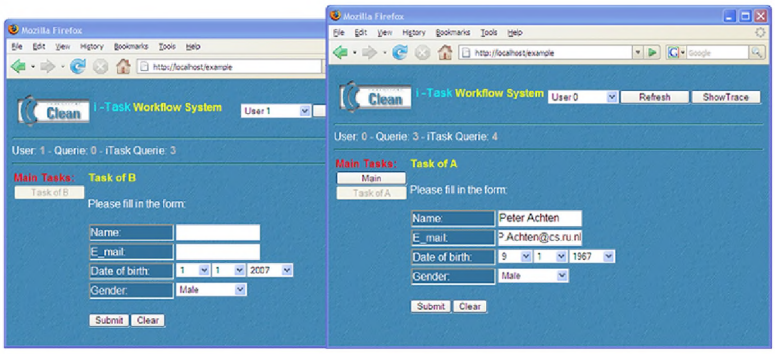


Figure 6.2: The case study with two workers: A (right window) and B (left window).

This case study is a representative example of a workflow situation. It has unpredictable execution times of tasks (as both workers can decide how much time to consume), non-locality (race between two workers and the outcome of worker B affects the tasks of worker A), and potential distribution of local work (both workers could perform the given task locally on the client).

Let `workflow` be the specification of the case study workflow. Every multi-user iTask program has the following preamble:

```
module admin                                                    1.
import StdEnv, StdiTasks                                        2.

Start world = multiUserTask [] workflow world                   3.
```

The main function is `Start` (line 3). `multiUserTask` generates a multi-user workflow infrastructure for the specification `workflow`.

In the case study worker A and B perform the same task, `person_admin`. We capture this pattern concisely by means of a parametrized workflow function `delegate` which is also parametrized with the worker identification values 0 (worker A) and 1 (worker B).

```
workflow = delegate 0 1 person_admin                            4.
```

```
delegate :: UserId UserId (a → Task a) → Task a | iData a                    5.
delegate userA userB taskf = a -||- b                                        6.
where                                                                        7.
    a = userA @: ("Task of A",taskf createDefault)                           8.
    b = userB @: ("Task of B",taskf createDefault                            9.
        >>= λrb → userA @: ("A checks B",taskf rb))                          10.
```

The function `delegate` specifies the main structure of the workflow as described above: two tasks (`a` and `b`) are created simultaneously (`-||-`). The first task is provided with an initial default value (`createDefault`), and this is the task that needs to be performed by worker A (line 8). The second task is provided with the same initial default value, and needs to be performed by worker B (line 9). When finished, worker B has produced `rb`, which is passed along via the monadic bind combinator `>>=` to worker A again, who can decide to work with `rb` (line 10).

The task `person_admin` that is performed by worker A and B double-checks filling in a personnel record of type `Person`:

```
:: Person = { name        :: String,   e_mail :: String                      11.
            , dateOfBirth :: HtmlDate, gender :: Gender }                    12.
:: Gender = Female | Male                                                    13.

person_admin :: Person → Task Person                                         14.
person_admin p = doubleCheck p checkPerson                                   15.

checkPerson :: Pred Person                                                   16.
checkPerson {name,e_mail}                                                    17.
 | name=="" = (False, [Txt "Please fill in your name"])                      18.
 | not ok    = (False, [Txt "Incorrect e-mail address"])                     19.
 | otherwise = (True,  [])                                                   20.
where                                                                        21.
  ok = not (isMember '@' (fromString e_mail)) || e_mail==""                  22.
```

The predicate `checkPerson` determines whether the worker did a good job. Double-checking a worker's output is also a parametrized workflow function:

```
doubleCheck :: a (Pred a) → Task a | iData a                                 23.
doubleCheck a p                                                              24.
= [Txt "Please fill in the form:"]                                           25.
  ?>> editTaskPred a p >>= λna →                                             26.
  chooseTask [ Txt "Received information:"                                   27.
             , toHtml na, Txt "Is it correct?" ]                             28.
             [ ("Yes", return_V na)                                          29.
             , ("No", doubleCheck p na)]                                     30.
```

(`doubleCheck a p`) uses (`editTaskPred a p`) (line 26) to generate a web form to enter a value of the type of `a`: again, the type class restriction guarantees that this is possible for the particular type of `a`. Once the worker has successfully entered a correct value, then this is passed monadically as `na` (line 26) to the next sub-task (lines 27-30): the value is displayed (`toHtml na`, line 28), and the same worker is asked to confirm whether she is sure about the information she has entered: if she confirms (line 29),

then doubleCheck returns that value, and if she declines (line 30), then doubleCheck *recurses* with the new value.

What remains to be done is to include 'boilerplate' code for deriving instances of the custom data types of the required generic functions:

```
derive gForm   Person, Gender  // create form                    31.
derive gUpd    Person, Gender  // process edit operation        32.
derive gPrint  Person, Gender  // serialize value               33.
derive gParse  Person, Gender  // deserialize value             34.
```

This completes the case study.

6.2.3 Opportunities for optimization

The case study illustrates a number of opportunities for efficient evaluation: in the current iTask implementation, every worker action triggers a round-trip between the client browser and server application. The actions of worker A and B are largely independent: still, the application takes both current states into account whenever either worker submits information. This can be improved if the system would restrict itself only to the required information. Also, one can imagine that the complete person_admin task can be executed on the client, without any communication with the server. This requires on-client evaluation of arbitrary Clean code. In the next section, we present an extension to the iTask system that allows the workflow engineer to specify these properties, while maintaining correct handling of multiple users and global effects.

6.3 Controlling the evaluation of tasks

In this section we introduce two annotations that the workflow engineer can use to control the behavior of any task t. The annotations are (UseAjax @>> t) and (OnClient @>> t), and are implemented as type class instances:

```
:: SubPage = UseAjax | OnClient

class (@>>) infixl 7 b :: b (Task a) → Task a | iData a
instance @>> SubPage
```

6.3.1 The "UseAjax" annotation

Modern web browsers support Ajax-technology. Ajax allows web applications to define *call-back functions* on the client in JavaScript. When a client browser submits a request for a new page to the server it usually receives a completely new page and renders the new page. Using Ajax, the call-back function handles the response of the server instead of the browser. This happens asynchronously, hence the user can continue to work on the page in the browser while the request is being processed. With this technique web pages can be updated partially, which results in a much

more responsive behavior resembling desktop applications. As we will see, the implementation can benefit from it too: in many cases only the effect of the particular task being performed has to be calculated, instead of all tasks.

The workflow engineer can annotate any task expression. This requires some consideration, because Ajax imposes a performance penalty. As a rule of thumb, worker tasks (tasks assigned to a worker with the `@:` operator) are suitable candidates for annotation because they clearly form a unit of work and they own graphical estate on the web page. To support this rule of thumb, the workflow engineer can set these tasks to "UseAjax" by default by setting a switch in the iTask library and hence readily create "Ajax threads" for explicit worker tasks. For some applications this default granularity might turn out to be too coarse grained. Using the `UseAjax` annotation allows the workflow engineer to create Ajax threads at any level. They may be invoked conditionally, they may be nested, and they may occur in recursive definitions.

The `UseAjax` facility is also a useful feature because it can serve as an automatic backup mechanism when client site evaluation is somehow not possible.

6.3.2 The "OnClient" annotation

In Sect. 6.2.3 we suggested that the double checking personnel data task can be executed completely on a client instead of the server. The only change to the case study specification is adding the appropriate `OnClient` annotations in the `delegate` function:

```
delegate :: UserId UserId (a → Task a) → Task a | iData a                    5.
delegate userA userB taskf = a -||- b                                        6.
where                                                                        7.
  a = userA@:("Task of A", OnClient @>> taskf createDefault)                 8.
  b = userB@:("Task of B", OnClient @>> taskf createDefault                  9.
      =>> λrb → userA@:("A checks B", OnClient @>> taskf rb))                10.
```

Any such annotated task is a "client thread", and is supposed to be executed in the client browser. Not every task can always be evaluated on the client. For instance, a task might inspect or change information in a database stored at the server side. Due to the non-locality of worker actions, their effect can only be determined with global knowledge of the state of worker tasks, which is only available on the server. Consequently, the `OnClient` annotation must be seen as a *wish*: if possible the task is evaluated on the client, but the evaluation strategy might be forced to do the work on the server. It is also possible that a client task is part of a larger task to be executed on the server. When the client task is finished one has to be able to switch back to the server for the continuation. Now we can appreciate the availability of the `UseAjax` annotation even more: whenever `OnClient` evaluation of a task is not possible we can simply change it into an Ajax call instead and execute the task on the server.

6.3.3 Discussion

With the two new annotations, `UseAjax` and `OnClient`, the workflow engineer can control the evaluation of tasks in a lightweight way. However, the implementation of these annotations is by no means lightweight because it needs to handle many issues. One issue is that for every worker event it needs to figure out which Ajax thread (if any) has to handle the event. This event may cause the associated task to terminate. In that case the Ajax thread has to terminate as well, and the parent thread has to be activated to determine the next tasks to deal with. This can result in a cascade of activated-terminated Ajax threads. Another issue is that, due to non-locality of worker actions, tasks may disappear and consequently also their associated Ajax threads. In these cases the evaluation strategy has to resort to the standard evaluation technique. Switching of evaluation strategy is also vital for those `OnClient` tasks for which it turns out that they cannot be evaluated on the client. Using the Ajax infrastructure allows the iTask toolkit to turn a failing `OnClient` task automatically into a `UseAjax` task, and hence have the task evaluated on the server. This can only be done if the server has either full knowledge of the states of all clients, or if it can completely reconstruct their state on demand. The iTask toolkit uses the latter strategy, which is explained in Sect. 6.4. After that, we show in Sect. 6.5 how Ajax technology is incorporated and client-side evaluation in Sect. 6.6.

6.4 Standard iTask Implementation

In order to appreciate the implementation of the new extensions to the iTask toolkit, we need to focus on its initial implementation. The material presented in this section is a revision of [PAK07].

6.4.1 A Functional Approach

As discussed earlier, the initial iTask toolkit creates thin-client web applications. This means that the client browsers are used for rendering purposes only. All events of all web clients that correspond with the workers that are currently using the workflow application are sent to a single server application. This server iTask application is executed whenever an event is received. The result is a new web page for the worker. This page depends only on the input event and the current state. The state is adapted by the iTask server application. A number of fundamental design decisions have been taken in the creation of the iTask toolkit. In a nutshell, these are:

1. There is a single declarative iTask specification such as the `admin` case study in Sect. 6.2 from which all code (including Html) is generated.

2. Task editors have persistent state. A task editor displays the state and allows the user to alter this state in such a way that only values of the same type can

be created. Rendering, editing, updating, and storing and retrieving values is all done generically.

3. An iTask program is a pure function, hence referentially transparent, and will produce the same result (and same effect) when applied to the same input (and state). More precisely, it will produce the same task editors in the same order.

An iTask application has an effect on its external world and keeps track of the various persistent storages. Its *state* is used for this purpose, and is discussed in Sect. 6.4.2. The evaluation of an iTask expression gives rise to the concept of a *Task Tree*, which is presented in Sect. 6.4.3. Finalized tasks are *rewritten* in an analogous way as graph reduction takes place in the implementation of functional languages. This is explained in Sect. 6.4.4.

6.4.2 The iTask State

To the workflow engineer, the task type (`Task a`) is opaque. Internally, it is a state transformer function of type:

```
:: Task a :== *TSt → (a,*TSt)
```

The state of an iTask application is the uniquely attributed `*TSt`. Every task is applied to a `*TSt` value, and returns a modified `*TSt` value, as well as the result of the work being performed, which is a value of type `a`. Although iTask applications are programmed in a monadic style, it is the underlying uniqueness typing of Clean that guarantees that the `*TSt` value is passed single threadedly from one iTask transition function to the other.

```
:: *TSt = { hst  :: *HSt,     activated :: Bool
          , html :: HtmlTree, params    :: TParams }
```

`*TSt` extends the uniquely typed iData state `*HSt` [PA06b]. For this paper, two components of `*HSt` are relevant. The first is an accumulator in which the state of all web form editors is collected, such that they can be saved in persistent memory when the iTask application terminates. The second is the `*World` environment value that allows it to perform these operations effectively.

The boolean value `activated` acts as a *control token* passed from one combinator to another indicating which tasks have terminated, which tasks are active, and which tasks need to be activated. When a combinator is called, `activated` tells it whether it has to be activated or not. If its value is `False` the task will not be activated at all and a (default) value of proper type is returned immediately, generated by making use of the generic machinery. Otherwise the task is activated and the combinator is applied on the current `*TSt` state, possibly activating other iTask combinators in turn. When the combinator is returning a result, the corresponding task may or may not be terminated. If the task is not terminated, the returned `activated` value is `False` and a (default) value is again returned as result of the task. If the task is completely terminated, `activated` is set, and the value returned is the final result of

the task. Since the task has ended, this result is passed on to the next task or set of tasks which in turn are activated as well. Hence, a combinator always returns a result of proper type, but only the values returned from finished iTask are meaningful and are passed on to other activated tasks. The meaningless default values are only passed around and never used.

The Html code generated by tasks is accumulated in the `html` field. The information for the intended worker is filtered out.

The remaining information is collected in the `TParams` record. This information is necessary to construct a `*TSt` value when needed.

```
:: TParams =   { userId      :: UserId
               , options     :: Options
               , taskNr      :: TaskNr }
:: TaskNr  :== [Int]
:: Options =   { tasklife    :: Lifespan
               , taskstorage :: StorageFormat
               , taskmode    :: Mode
               , gc          :: GarbageCollect }
```

The `userId` is the unique identification of the worker who has to perform the corresponding task. An iTask can have many options which are stored in the `options` field. For instance, the `Lifespan` option defines in which memory (in the web page on the client side, or on the server side in a relational database or in a file on disk) the status of the task is stored when the application ends. Last but not least, every iTask obtains a unique identification, for which the `tasknr` field in the `*TSt` state is used. Such a unique identifier is crucial in order to retrieve the iTask state information from the different persistent stores. Tasks are numbered dynamically, in the same way as chapters, sections and subsections are numbered in a book or in this paper: tasks on the same level are numbered subsequently, whereas a subtask `j` of task `i` is numbered `i.j`. Task numbering allows us to determine how tasks are related to each other. Just by looking at the task numbers we can figure out the ancestors of a task and which subtasks it has spawned. In the standard iTask implementation this knowledge is used for garbage collection of subtasks. We can now use it conveniently for our new annotations to determine which (parent) thread to activate when an event has occurred.

6.4.3 The Task Tree

An iTask application remembers its point of evaluation. In a language like C or Java the point of evaluation is remembered by using a stack. For iTask it is better to use a tree, the so-called *Task Tree*. The reason is that we are dealing with a multi user system: people can work on many tasks simultaneously. As a matter of fact, also one user can have several tasks she can work on at the same time. At any time we have to be able to administrate the progress made on any task by any worker. Furthermore we have seen that new tasks can be created while other existing tasks might not be needed anymore.

A tree structure is well suited for the administration of all this. Each iTask may depend on other iTask and finally on basic iTask editors. The dependencies are determined completely by specified iTask combinators. The used combinators form the nodes in the *Task Tree*, the basic editors the leaves.

The contents of a *Task Tree* varies over time. An activated task might be changed into a finished task, new tasks can appear and complete old sub trees may be pruned because the corresponding tasks are no longer needed.

Fig. 6.3 depicts a snapshot of the Task Tree of the `admin` case study. User A (id 0) and user B (id 1) are working on their `person_admin` task. The user id (in the left upper corner), the iTask combinator name, and the task number are displayed in each node. User 0 finished an `editTaskPred` and is now working on a `chooseTask`. User 1 is still working on an `editTaskPred`. There are two threads created, one for each `person_admin` task. The grey area indicates which combinators belong to which thread.

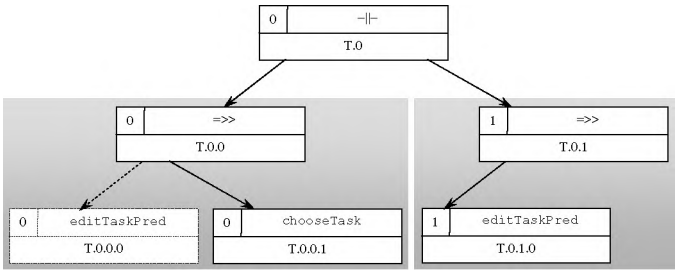


Figure 6.3: The Task Tree at work with the `admin` case study.

Although an iTask application can remember its previous point of evaluation, it is not realized by interrupting a running Clean application, waiting for the next event received from some user, and continuing execution as one would do in a C implementation. The reason is that there is not a single point of execution, there are several: all active worker tasks. So, a parallel evaluation order of the iTask specification would be appropriate, which is quite different from the normal order evaluation used in Clean. Implementing a parallel evaluation strategy is challenging and time-consuming. It turns out that there is an elegant, much simpler technique that achieves the same result. We exploit the fact that we are working with a *purely* functional language: the result of a function only depends on its arguments. We also make use of the iData library: every editor ever being used automatically stores its state in its specified (persistent) memory and this state is automatically recovered when the editor is activated again (see [PA06b]). Reconstruction of the previous point of evaluation is accomplished by re-evaluation of the program with the new input received where at the same time the effect of old inputs is recovered automatically thanks to the iData being used.

Consequently, the Task Tree does not really exist: part of it is reconstructed via the re-evaluation of iTask combinators (they are just plain Clean functions), part of it is reconstructed using the stored information of the iData editors. In this way the Task Tree is reconstructed from scratch whenever an event is received. This

technique is also very suited for dealing in a robust way with a complicated, hard to control, distributed environment such as the internet.

On demand, the iTask application can show the contents of this virtual *Task Tree* to the user. In this way one can get an overview of who is doing what, which tasks are finished and what their results are.

6.4.4 Global Task Rewriting

Mature workflow systems should run for years and are used by many workers. This implies that the *Task Tree* as described above would grow indefinitely over time. For a real world workflow application this is of course unacceptable. The evaluation of an iTask is therefore optimized in a similar way as a function application is optimized in any implementation of a functional language: when a function has been evaluated, the function call is replaced by its result. Similarly, when a task is finished, it is replaced by its result. This is noticeable in the *Task Tree* as well: a combinator node in the *Task Tree* is replaced by the resulting task value. This *Global Task Rewriting* increases efficiency because *Task Trees* can be reconstructed much faster. Although not discussed in this paper, the iTask toolkit has iterative combinators such as `foreverTask` that repeat tasks infinitely many times. These can restart from scratch and even reuse the task numbers. In this way both the *Task Tree* and the task numbers have proper upper bounds (in size).

The downside is that the implementation becomes more complicated as well. The iTask information stored in the persistent memories needs to be garbage collected which is not always trivial. As we will see, *Task-Tree* rewriting also has an impact on the implementation of our new annotations.

6.5 Implementing Ajax calls via Local Task Rewriting

In this section we show how Ajax-threads are incorporated within the iTask toolkit. The key idea of Ajax is to enable JavaScript fragments that reside in a web page to engage in asynchronous communication with servers, a functionality that was strictly reserved to browsers before Ajax came along. The result of this technology is that one can create web pages that are constructed out of arbitrarily many 'classic' components (i.e. go along with the browser-server communication cycle) as well as arbitrarily many components that handle their private content with servers of their choice. In order to set up the asynchronous communication, a JavaScript creates a so-called *XML http request* object. With that object, it can communicate with any server of its choice. Usually, this communication is asynchronous, but synchronous communication is also possible. In case of asynchronous communication, a *callback function* is associated with the *XML http request* object, by overriding its *onreadystatechange* method. This function is called whenever the server has responded, and it will find its result in the *responseXML* data member of the *XML http request* object.

If we want to use this technology within the iTask framework we will need to create and store the proper callback functions. These callback functions will necessarily update the *Task Tree locally* instead of globally as described in Sect. 6.4. We make use of the property that a *Task Tree* cannot only be reconstructed from the root of the tree: any subtree can be reconstructed in a similar way as well. The reason is that due to referential transparancy, the same *Task Tree* will be reconstructed, and this property also holds for any subtree. So, we can reconstruct and rewrite the *Task Tree locally*, i.e. starting from any node in the tree if only we can store and determine the callback function that handles this part of the tree. This is discussed in Sect. 6.5.1. Due to possible non-local effects of worker tasks, we may need to switch between local *Task Tree* rewriting and global *Task-Tree* rewriting. This is described in Sect. 6.5.2. Finally, we discuss what has been achieved after this step in Sect. 6.5.3.

6.5.1 Thread Storage and Creation

Every subtree of the *Task Tree* has been created by one of the iTask combinators (see also Fig. 6.1). To reconstruct a subtree we have to know which iTask combinator (thread) is responsible for its construction and we need to know with which arguments this function has been called. This information has to be stored somewhere such that we can re-evaluate the function later, as a special kind of callback function. So, we must be able to store and retrieve closures. Clean already has powerful means for doing that. By using Dynamics, any type, including function types, can type safely be stored and even be exchanged between independently programmed Clean applications [Wee07]. Exchange of dynamics between two applications requires the presence of a dynamic linker. Loading dynamic code and data with a linker consumes a significant amount of time. Because we are dealing with one and the same server application, this is not necessary. We only make use of Clean's ability to serialize and de-serialize functions. The two functions that do this are `serializeClean` and `deserializeClean`:

```
serializeClean   :: (Task a) → CleanSerialization
deserializeClean :: CleanSerialization → Task a

:: CleanSerialization :== String
```

Note that type correctness is no longer automatically guaranteed, so our storage administration should better be correct, which is assured by the way they are created and used.

```
:: ThreadTable :== [TaskThread]
:: TaskThread  =   { thrCallback :: CleanSerialization
                   , thrParams   :: TParams }
insertNewThread   :: TaskThread *TSt → *TSt
deleteThreads     :: TaskNr     *TSt → *TSt
findThreadInTable :: TaskNr     *TSt → (Maybe TaskThread,*TSt)
```

In the ThreadTable threads are stored in a record structure of type TaskThread. The serialized iTask combinator is stored in the field thrCallback. In order to reconstruct the *TSt value on which the combinator function has to be applied, we also store the TParams information, which contains the appropriate options, userId, and tasknr. There is no need to store the activated token nor the accumulated html because this information gets accumulated from nodes above the subtree.

```
mkTaskThread :: (Task a) → Task a
mkTaskThread task = storeAndEvalThread
where storeAndEvalThread tst=:{activated,params}
      = case findThreadInTable params.tasknr tst of
          (Nothing,tst)  = storeAndEvalThread (insertNewThread
                           { thrParams   = params
                           , thrCallback = serializeClean task
                           } tst)
          (Just thr,tst) = evalTaskThread thr tst
```

For every task annotated with UseAjax, mkTaskThread is called. It stores the corresponding task, a state transition function, in the table if this has not been done in a previous incarnation (note that also this code might be re-evaluated several times). Finally it evaluates the task thread by calling evalTaskThread.

```
evalTaskThread :: TaskThread → Task a                                          1.
evalTaskThread {thrParams,thrCallback} = evalTask                              2.
where                                                                          3.
  evalTask tst=:{params,html}                                                  4.
  # (a,tst=:{activated,html=nhtml})                                            5.
              = deserializeClean thrCallback                                   6.
                {tst & params = thrParams, html = noHtml}                      7.
  | activated                                                                  8.
    = (a,{deleteThreads thrTaskNr tst & params = params})                      9.
  | otherwise                                                                  10.
    # newhtml = DivCode (showTaskNr thrParams.taskNr) nhtml                    11.
    = (a,{tst & params = params, html = html +|+ newhtml})                     12.
```

The function evalTaskThread can reconstruct the desired subtree of the *Task Tree*. It is crucial to observe that this function can be called in any context. Therefore we can use it to regenerate the subtree when an Ajax call is done. In that case one first has to determine which thread from the ThreadTable should be selected. This is explained in Sect. 6.5.2.

The function evalTaskThread de-serializes the stored iTask combinator and reconstructs the iTask TSt state such that the proper subtree is reconstructed (lines 6-7). When the combinator task is finished (lines 8-9) the thread removes itself from the thread table. If the thread is not finished, more work on it has to be done in the future. The new Html code generated by the thread, nhtml, is appended to the HTML accumulator html marked by an Html Div construct that is labelled with the task number of the thread. This enables the JavaScript callback function on the client side to replace the old HTML code (which is labeled with the same task number) with the new one, leaving all other code unchanged. Therefore the part of the page

that is updated depends on the chosen thread.

6.5.2 Determining which Threads to Activate Given an Event

Using callback functions for handling events is a common technique. However, in this case we cannot assign callback functions for an event beforehand because we deal with a distributed multi-user web enabled system. Due to global effects, a once constructed subtree might not even exist any-more. As a matter of fact, when a task is no longer needed, all its administration is removed. Also its threads are removed from the thread table. We have to determine dynamically which thread is able to handle an event, if any. How can we do this?

The form committed by a user has been created by an iTask editor. Each iTask has a unique number, so we can encode this number in the event. The numbering discipline (Sect. 6.4.2) allows us to determine which thread to activate.

Assume that no global effects occurred. In the thread table we search for the ancestor thread that is most closely related to the event: a task with the same prefix number. If such a thread can be found, and the corresponding task is indeed assigned to this particular user, it is evaluated. The subtree is reconstructed as described above and this subtree includes the basic iTask corresponding to the event. This task can handle the event as usual. If afterwards the chosen thread task is not finished yet, the corresponding Html code is communicated to the client (6.5.1) where the JavaScript callback function uses it to update the corresponding area on the web page. If the thread is finished, it removes itself from the table (6.5.1). Termination of this thread can trigger the evaluation of the next thread in the workflow structure. We search again in the thread table to find an enclosed thread which is now most closely related to the event and activate it. This process can repeat itself several times. Eventually, the page area that gets updated depends on the last thread activated in this way.

Assume that global effects did occur. How can we find out what has happened? We can find it out by reconstructing the whole *Task Tree* because this gives us the exact status of all worker tasks, but that is exactly what we wanted to avoid in the first place. Instead, for every worker we maintain an administration of type `GlobalEffect`, in which we keep track of global effects:

```
:: GlobalEffect = { versionNr      :: Int
                  , newThread      :: Bool
                  , deletedThreads :: [TaskNr] }
```

If a thread has become obsolete due to an action of another worker, its task number is added to the administration (`deletedThreads`) of the worker of that task. If a new task is assigned to a specific worker, this fact is administrated in `newThread` as well. Also a version number is administrated for proper handling of browser buttons and cloning of windows.

The `GlobalEffect` administration is inspected before threads are determined. If there is a new thread, or if a thread has been deleted related to the event, we fall back

to the old way and reconstruct the *Task Tree* starting from the root and construct a whole new page. Otherwise we can start looking for the right thread as described earlier. As a result, one cannot predict which part of a page will be updated. It can vary from a small area exactly covered by the closest thread, or a bigger area, or ultimately even the entire page.

6.5.3 Discussion

In this section we have shown how to incorporate Ajax threads in the iTask toolkit. As we have explained in Sect. 6.3, we have chosen to turn every worker task (i.e. a task assigned to a worker with the `@:` function) into an Ajax thread. As a result, the page that is displayed to a worker consists of a set of tasks, each of which can be handled individually by the worker without the need to wait for the full page to reload. The latter is only necessary in case her action has caused a non-local effect. In this way, the user experiences a smoothly operating workflow application. The workflow engineer can further fine-tune the workflow application by adding `UseAjax` annotations in the right places.

6.6 Implementing Local Task Rewriting on the Client

The contributions to the iTask toolkit described so far still result in a *thin-client* architecture: web browsers are used for rendering purposes, and all computations take place on the server. Any iTask that does not require server-side database or file access can in principle be evaluated on the client instead of on the server. In this section we describe how this can incorporated within the iTask toolkit. Because task expressions are full-fledged Clean functions, and the iTask toolkit is based on generics, this means that non-trivial Clean code needs to run in a browser, which is something new. In Sect. 6.6.1 we show how we have done this by compiling an iTask program to two images. One image runs on the server and is a Clean executable, and one image runs as an *interpreted program* on every client. The two images run the same program, and reconstruct the Task Tree as described in the previous sections. Hence, also for the interpreted image callback functions need to be created and stored. This is described in Sect. 6.6.2. Again, non-local effects need to be taken into account. This is explained in Sect. 6.6.3. Finally, we discuss the achievements in Sect. 6.6.4.

6.6.1 Client-Side Evaluation of Clean Code

The clients need to execute iTask expressions, which can be arbitrarily complex Clean expressions. One may choose to create a *plug-in* for Clean applications for these client browsers, but this conflicts with our design decision to implement iTask as much as possible using existing web technology. Instead, we have chosen to make use of the Sapl interpreter [JKP06] (chapter 3). Sapl is a very simple functional

language in which only functions appear: it has no data structures, and no pattern matching. Due to its simplicity, the Sapl interpreter is very small. Despite its simplicity, it is faster than interpreters like GHCi, Helium, Amanda and Hugs, albeit not as fast as the code generated by the Clean or GHC compiler (for more details see [JKP06] or chapter 3). Being small and relatively fast it is a well-suited candidate to incorporate in a browser as a Java applet.

In order to make the Sapl interpreter suitable for the web, it had to be re-implemented in Java (the original version was encoded in C). Another crucial step is to create a compiler from Clean to Sapl. This has been done, and we present the details of this work elsewhere. The result is that we can compile a complete Clean iTask application to Sapl.

Every iTask application is now compiled to two images: one compiled by the Clean system to native Intel code running on the server, one on the client which is interpreted by Sapl. The advantage of this approach is that we obtain two, almost identical, images of the *same* iTask application between which we can switch. The code generated for the client differs slightly from the code generated for the server: the client cannot deal with global effects and will act differently in these situations. This happens under the hood: the workflow engineer is not concerned with these aspects. The two images are generated from one and the same iTask specification. The Sapl interpreter and Sapl code are loaded by the browser once when a worker visits the workflow page for the first time. In addition, a *single, generic* JavaScript, independent of the iTask application, is loaded as well that handles *all* Ajax communication. This overhead is paid for only once.

6.6.2 Thread Storage and Creation

Now, whenever an `OnClient` annotation is specified for a task running either on the server or on the client, a modified `mkTaskThread` (Sect. 6.5.1) is called. The difference is that, when the `OnClient` annotation is encountered, not only a serialized version of the thread is made which can be executed on the server, but now also a serialized version of the thread is made which can be executed on the client. These two encodings are completely different though, due to the fact that we are dealing with two completely different implementations. So we need special conversion functions in Clean (and in Sapl) which can create the required serialization for Sapl (`serializeSapl` and `deserializeSapl`). To store the serialized client thread the thread table is extended with the field `thrCallbackClient`. The thread table is stored on the server as part of the state, and a copy of the relevant information of the thread table (only the threads intended for the particular client) is stored on the client as well.

```
serializeSapl   :: (Task a) → SaplSerialization
deserializeSapl :: SaplSerialization → Task a

:: TaskThread
  = { ... thrCallbackClient :: SaplSerialization ... }
```

If a client thread is created, the function `storeAndEvalThread` in `mkTaskThread` now additionally stores the serialized client thread for handling in Sapl. Furthermore, it

sets the `tasklife` field in the `options` of the `*TSt` task state to the option `Client`. The `tasklife` field indicates where task information should be stored. The administration of the iTask that should run on the client are stored in the browser page on the client. The setting propagates via this state to all subtasks being created by the thread. Hence for all subtasks it can be determined whether they should preferably run on the client or not. In the generated Html forms this knowledge is used in the encoding of the events.

6.6.3 Determining which Threads to Activate Given an Event

Initially, the evaluation of an iTask application starts on the server. It generates the first page containing the initial forms. The Sapl interpreter and Sapl code are loaded as a side-effect. When an event is generated, it is inspected on the client. There is one single special JavaScript script running for this purpose on the client side. This script operates as a switch: if the event is intended for the client, the script sends the event to the Sapl interpreter running as a Java applet. Otherwise the script sends the event to the server as if an ordinary `UseAjax` annotation was encountered.

The client basically performs the same actions as the server. However, the client cannot deal with global changes, or persistent storage handling on the server (e.g. database access). The general recipe is: in case of panic stop the execution on the client and fall back to the server-side handling of the event.

There are two types of global effects: effects caused by the client that have an effect on co-workers. The client can recognize this situation and take the panic exit. Vice versa, co-workers can also cause a global effect that affect the client who is ignorant of these facts, for instance when it has not connected to the server for a long time. To catch this situation each client periodically has to ask the server whether global effects are stored for him in the `GlobalEffect` record (Sect. 6.5.2). The Ajax callback technology is ideal for handling this situation. In the case of global effects a Boolean value can be set in the client such that the next event will be forced to communicate with the server such that client and server administration can be synchronized.

6.6.4 Discussion

With the `OnClient` annotation workflow engineers can create workflow systems that are able to compute arbitrarily complex computations on clients. This can significantly reduce the traditional round-trip communication between clients and server, it will reduce the workload of server applications, and it will enhance the worker experience as the workflow can respond quicker than in the old setting. Every workflow system has to be aware of non-local effects in any kind of implementation, and the iTask toolkit automatically detects whether this is the case and global Task-Tree rewriting is required. The workflow engineer immediately profits from this approach because she does not need to concern herself with the question what parts of the local computations must take place on the client and what parts must be done on

the server. The iTask toolkit automatically switches to global Task-Tree rewriting in case a client task performs such work.

6.7 Related Work

The new iTask toolkit as described in this paper allows high level specification of multi-user workflows. Forms are generated generically from type information, which considerably decreases the amount of HTML programming. Complex dynamic workflows can be created that can be evaluated at both the client and the server without any restriction. Actions of workers can safely affect that of co-workers: the tool-kit supports multi-user programming smoothly. The system is robust: computations that cannot be evaluated at the client side can always, and safely, be evaluated at the server side. We are not aware of any other functional system that has these features. However, there are functional approaches for handling web pages.

Links [CLWY06] and its recent extension formlets [CLWY07] is a functional language based web programming language. Links compiles to JavaScript for rendering HTML pages, and SQL to communicate with a back-end database. A Links program stores its session state at the client side. In a Links program, the keywords `client` and `server` force a top-level function to be executed at the client or server respectively. In Links processes can be spawned, and these processes can communicate via message passing. Client-server communication is implemented using Ajax technology. In iTask processes are not created explicitly as in Links programs. The novel `UseAjax` and `OnClient` are similar to Links functionality, except that we do not limit their use to top-level functions, but instead allow any (nested) task to be annotated. In the iData and iTask tool-kits, forms are generated generically for every data type, whereas in Links and formlets these need to be coded by the programmer. Links and formlets are designed for form based web applications, as is the iData toolkit, whereas the iTask toolkit extends this with multi-user workflow, including recursive, higher-order workflows.

Another functional language based web programming language is Hop [SGL06, LS07]. Like Links, Hop is compiled to JavaScript. It implements a strict separation between programming the user interface and the logic of an application. The main computation runs on the server, and the GUI runs on the client(s). These components can invoke each other (from GUI client to server via function calls, from server to client via signalling events). In particular the latter feature increases the expressive power of web applications, which are usually driven by the browser client side. Hop is a stratified language: each component is programmed in either one stratum in order to prevent it from performing operations that are considered to be illegal (e.g. database access by the GUI client, or rendering operations by the server application). Annotations control what stratum is used: within the main stratum (the server application code) ~ escapes to the GUI stratum, and within the GUI stratum one uses $ to escape to the server. Additional server logic can be invoked as a Hop service, which makes the design very modular. This has been implemented with Ajax. In the iData and iTask toolkits, we do not require a stratified language

approach to divide our attention to GUI programming versus application logic, because the GUI is mainly generated generically. The novel iTask annotations `UseAjax` and `OnClient` also allow fine-grained tuning of application logic, whereas an "exit strategy" is always available by relying on *Global Task Rewriting* strategy.

The Flapjax language [Kri07] is an implementation of functional reactive programming in JavaScript. Many of its features are comparable with those of Hop, and indeed both are designed to create intricate web applications. The main difference with our approach is that the iTask system is geared for distributed, multi-user, workflow systems in which the coordination and interaction of work is defined in a highly declarative style.

The enabling technology of client-side evaluation in the iTask tool-kit is SapI. We have seen that the complete Clean application is compiled to SapI. This enables our approach to use the full expressive power of Clean to perform intricate computations at the client side. A much more restricted approach has been implemented in Curry [Han07]: only a very restricted subset of Curry is translated to JavaScript to handle client side verification code fragments only.

6.8 Conclusions

In this paper we have presented a number of contributions to the iTask toolkit, a combinator library written in Clean to create workflow systems that run on the web in a pure functional style. The contributions to the workflow engineer are that she can annotate arbitrary task structures with two annotations: `UseAjax` and `OnClient`. Task structures annotated with `UseAjax` can be handled much more efficiently by the toolkit by using the underlying Ajax technology. Using this Ajax technology only the part of the web page corresponding to the task that is changed is updated instead of the entire web page. Task structures annotated with `OnClient` can be evaluated completely on the client side. This requires the Ajax technology since the client-side evaluation of a single task only updates that task and hence only a fragment of the web page.

The actual gain in efficiency cannot be predicted because it highly depends on the kind of workflow being specified. The standard evaluation strategy of iTask is already reasonably efficient thanks to global task rewriting. Local task rewriting is more efficient when there are no global effects. Otherwise local task rewriting will not be more efficient, the use of Ajax even introduces some minor additional run-time overhead. Local task rewriting will be more efficient than global task rewriting when the workflow is large, there are many users, and global effects occur occasionally.

Client-side evaluation by SapI has as advantage that internet traffic is avoided and that server processing load is relieved, but as disadvantage that the SapI interpreter is slower than the compiled code generated by the Clean compiler. For arithmetical operations Clean can be an order of magnitude faster, but when higher order functions are being calculated SapI performs quite well. Whether it is better to calculate on the client therefore depends on the size of the task, the kind of com-

putations being performed, the amount of traffic on the internet, the speed of the network, the speed of the server, and the speed of the client. For small tasks the overhead of interpretation on the client usually outweighs the communication delay, even on slow client machines with fast internet connections. It is up to the workflow engineer to make the optimal choices. In any case, the advantage of both the `UseAjax` and `OnClient` annotation is the elimination of blank browser windows when waiting for a new page.

The technical contributions are the incorporation of Ajax technology within the iTask tool-kit, the ability to convert any Clean expression to a Sapl function call, the introduction of Task-Tree rewriting strategies that can automatically switch when required by the tasks that they evaluate, and rearranging the architecture of the iTask tool-kit to incorporate these changes. We have maintained the declarative approach of the iTask tool-kit. Everything is generated from an annotated, single source specification with a low burden on the workflow designer because the system itself switches automatically between client and server-side evaluation when this is necessary without any effort of the workflow engineer. The iTask system integrates all mentioned technologies in a truly transparent and declarative way.

Chapter 7

iEditors: Extending iTask with Interactive Plug-ins

[1] ***Abstract*** The iTask library of Clean enables the user to specify web-enabled workflow systems on a high level of abstraction. Details like client-server communication, storage and retrieval of state information, HTML generation, and web form handling are all handled automatically.

Using only standard HTML web browser elements also has a disadvantage: it does not offer the same level of interaction as we are used to from desktop applications. Browser plug-ins can fill this gap. They make it possible to extend web-applications with interactive functionality like the making of drawings. In this paper we explain how plug-ins can be nicely integrated in the iTask system. A special feature of the integration is the possibility for a plug-in to use Clean functions as call-back mechanism for the handling of events. These call-backs can be handled on the server as well as on the client. As a result we are now able to create interactive iTask applications (iEditors) using plug-ins like graphical editors. Although complicated, distributed multi-user applications can be created in this way, reasoning about the program remains easy since all code is generated from one and the same source: the high-level iTask specification in Clean.

7.1 Introduction

The internet has become an important platform for the deployment of applications. Despite this popularity, for an application programmer it is still hard to write web applications. To overcome this, the iData [PA06b] and iTask [PAK07] toolkits have been developed. They enable the development of web applications at a high level of abstraction, where the programmer can focus on the essence of the application without having to deal with web details like HTML generation and client-server communication. An iData application automatically generates output (HTML) and automatically handles user changes made in an HTML form. The iTask system adds the concept of tasks to iData. An iTask application can be considered as a structured collection of tasks to be performed by one or more users. In iTask specifications the

[1] Originally published as [JPK09]

flow of control and information between tasks can be expressed. To enhance the performance of iTask applications, the possibility to handle tasks at the client side of a web application was added. For this the Sapl interpreter [PJKA08] (chapter 6) was extended to a full Clean interpreter [JKP06] (chapter 3).

iData and iTask make use of standard HTML elements. In many cases these standard elements do not suffice for the creation of desktop-like applications. Browser plug-ins can be used to overcome this. Examples of plug-ins are media players for playing music and movies and Java Applets that offer the possibility to run Java programs at the client side of web applications.

When developing a web application using a plug-in the programmer has to deal with the following issues:

1. How to include the plug-in in the web application?
2. How to load relevant data into the plug-in?
3. How to transfer relevant data from the plug-in to the server application?
4. How to do specific processing for the plug-in (e.g, event handling for editors)?

For the inclusion of plug-ins in web applications, standard solutions in HTML exist. The other issues are mostly handled on an ad hoc basis, depending on the kind of application developed.

In this paper we focus on a more systematic solution for the last three issues. The focus is on the inclusion of Java Applet plug-ins [GJS96, Sun08] into iTask applications using generic [Hin00] programming techniques. The presented techniques are not restricted to Java Applets alone but can also be used for communication with other kinds of plug-ins like advanced text editors (e.g. fckeditor [Kna03]). For incorporating plug-ins into iTasks, a generic (read: poly-typical) framework is developed. The benefits for an application programmer are:

- A plug-in can be used with a minimum of programming effort and use of specific interface code. Generic functions take care of the conversion of Clean to Java data and back. They also take care of the communication between web-application and plug-in;
- One can define call-backs for the plug-in in Clean which can be handled either on server or client. Server handling can be used for executing more time consuming functions and client handling can be used for events requiring a quick response like mouse-event handling; For client-side evaluation of call-backs the Sapl interpreter is used;
- Plug-in tasks behave like ordinary iTasks. If a suitable plug-in already exists, the application programmer only has to define Clean types (matching the content and event types of the plug-in), similarly to those for ordinary iTasks. In order to include a plug-in into an iTask application only two interface functions are needed. For Java Applets the interface with plug-ins is encapsulated into a generic Java class.

We will call an iTask plug-in with Clean call-backs an iEditor. The technical contributions are:

- The seamless integration of plug-in tasks in the iTask formalism. This is realized by specializing the generic HTML generation and data update functions, in a completely transparent way for the application programmer;
- The use of Clean and Sapl dynamics [Wee07] for realizing fine grained control over call-back function handling. Clean expressions are serialized at the server side, moved to the client side and executed there (this requires a referential transparent formalism). In this way it is possible to move entire computations from server to client in a dynamic way;
- A generic way to exchange data between Clean and Java. On the Clean side this is realized by standard generic print and parse functions. On the Java side this is realized by the Java reflection [McC98] mechanism.

The structure of this paper is as follows. In Section 7.2 we start with a short survey of the iTask system and architecture. In Section 7.3 we discuss the issues to be dealt with for including plug-ins in iTask and we give an example of the use of iEditors. Section 7.4 discusses the implementation of iEditors. In Section 7.5 we present a generic framework for the exchange of data between Clean and Java. In Section 7.6 we discuss some alternative uses and implementations of the techniques we developed. Section 7.7 compares our solution with other approaches that use client-side processing. Finally, we end with some concluding remarks in Section 7.8.

7.2 The iTask toolkit

The iTask toolkit [PAK07] is a web-based combinator library written in the lazy, purely functional programming language Clean. It can be used to implement powerful web-applications like online shops, etc. We briefly repeat the most important characteristics of iTask. A task in iTask can be a basic task or a combination of tasks:

- A basic task is created by the `editTask` function, which turns an element of an arbitrary data type into an editable web form. User edits of the form lead to automatic updates of the underlying data type;
- Task combinators enable the combination of tasks. Combinators are used to control the flow of processing and data from one task to another. Tasks can be performed sequentially, in parallel and distributed over several users. New tasks can dynamically depend on the results of previous tasks.

In the original iTask architecture all processing is done at the server side of the application and all user actions lead to a complete update of the web-page the user is editing. In [PJKA08] (chapter 6) we showed how we can update sub-tasks in web pages and reduce the overhead of client-server communication in iTask applications

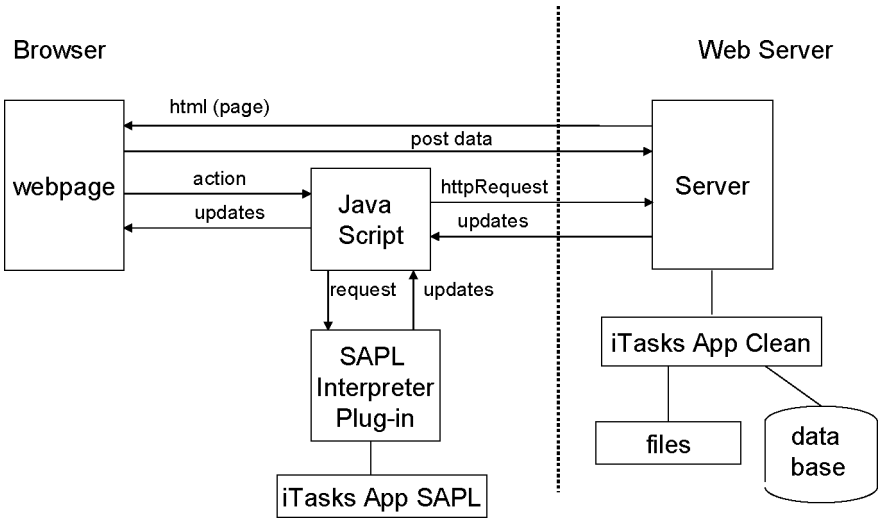


Figure 7.1: The architecture of an iTask Application

by adding Ajax [Gar05] and client-side evaluation of tasks. The plug-in extensions discussed in this paper extend the set of basic tasks with powerful interactive tasks. This is done in a way that does not restrict the way in which tasks can be combined with combinators.

7.2.1 The Architecture of iTask Applications

The architecture (see Fig. 7.1) of iTask applications is representative for web applications based on the Ajax philosophy (web 2.0 applications [Gar05, Pau05], [W3 08] gives details about web development with Ajax). It has the following characteristics:

- An iTask application consists of two images. A server executable running in native code at the server side of the application and a client-side image running in the Sapl (Simple Application Programming Language) interpreter that is integrated in the web browser as a plug-in;
- Both the server and client images are generated from one single source programmed in Clean. From this source the server executable and a client Sapl program are generated by the Clean compiler. Both the Clean executable and the Sapl source comprise the complete iTask program. Tasks can be handled either at the server or the client. In principle, it is even possible to run the complete application (all tasks) at the client, except for the storage and retrieval of information in files and data bases;
- The server application initially generates a complete HTML page (web form) that is displayed in the client browser;
- User actions in the web form can be handled as normal post messages by the server or as an `httpRequest` by either client or server. In the first case

a complete new HTML page is generated. In the second case it should be decided in JavaScript whether a request to either server or client application must be made. As result of the request a (partial) update of the web-page is made;

- The JavaScript at the client side is generic (the same for all iTask programs). JavaScript acts as an intermediary between client and server and client and SapI interpreter. It takes care of updating the page with results from the server or client and it transforms user actions in the forms into calls for server or client application.

The use of JavaScript is a characteristic of all Ajax-based applications, but in our case the JavaScript functions are only a means for passing requests and results between the server application, client application and web-page. All application related programming is done in Clean. In this paper we extend this architecture with plug-in communication.

7.2.2 The SapI Interpreter and Clean-SAPL dynamics

To execute tasks and Clean functions at the client-side, we need a Clean platform there. This is realized by making a plug-in version of the SapI interpreter [JKP06] (chapter 3) and a Clean to SapI compiler. By using a Java Applet for the interpreter, client-side Clean processing becomes available for all major internet browsers. The interpreter, originally realized in C, was re-implemented as a Java Applet with a performance penalty of less than 40%. This means that this interpreter is still considerably faster than other interpreters like GHCi, Helium and Hugs (see [JKP06] or chapter 3). We also constructed a Clean to SapI compiler, supporting the full Clean language. The generated SapI code can be loaded into the SapI interpreter at start-up of web applications. Loading times of SapI and client program (excluding the time needed to load the Java virtual machine) are comparable to that of web pages including JavaScripts of about 1000 lines.

A special feature of the SapI interpreter is that we can use a dedicated form of Clean dynamics [Wee07] for it. With dynamics it is possible to serialize a Clean expression (closure) to a string, store the string somewhere, retrieve the string at a later moment, turn it into a Clean expression again and execute it. We extended the dynamics features of Clean in such a way that it is also possible to serialize an expression in a Clean executable and de-serialize it in the SapI interpreter (running the corresponding SapI program), and execute the expression there. This is a powerful feature because it makes it possible to migrate execution of a Clean program from server to client. In this paper we use this feature for executing call-back functions at the client side.

7.2.3 Examples of iTask Applications

To give an idea of the iTask system, we give some small examples. Creating a basic task in iTask is simple. With the `editTask` function one can turn an element of

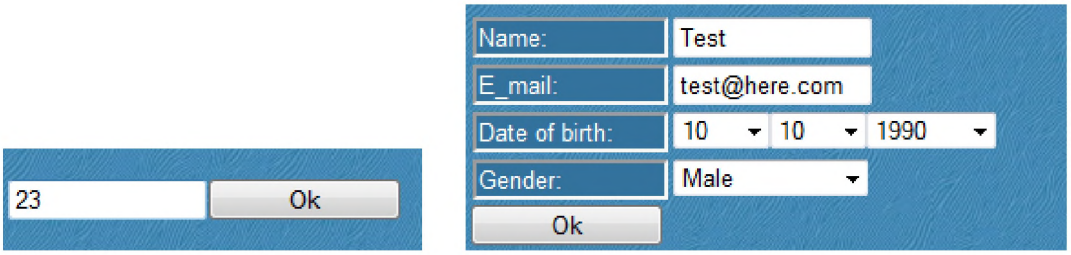


Figure 7.2: editTask for Int (left) and Person (right)

an arbitrary data type into a task. As a result an editor for the data type element is created residing in a web form. A user edit action of this form results in an automatic update of the data type that can be further processed by the remainder of the iTask application. `editTask` has two arguments: the name of the button that the user should press to end the task and the initial value of the editor. Here two examples of the use of this function are given: `simpleInt` creates an editor for an integer while `simplePerson` creates an editor for an element of type `Person`. We also give the definition of the type `Person`.

```
simpleInt :: Task Int
simpleInt = editTask "Ok" createDefault

:: Person = { name        :: String
            , e_mail      :: String
            , dateOfBirth :: HtmlDate
            , gender      :: Gender
            }
:: Gender = Female | Male

simplePerson :: Task Person
simplePerson = editTask "Ok" createDefault
```

Fig. 7.2 shows the resulting editors created when respectively `simpleInt` and `simplePerson` are called. Note we use `createDefault` for the initial value of the editors. The fields in the form now get default values generated by the system using generic functions.

The 'simple' examples just create a form to be filled in by a single user, yielding a value of the corresponding type. In the following example a combinator is used to let two users perform tasks after each other:

```
addMultiUserTask :: Task Int
addMultiUserTask
=           0  @:: editTask "Ready" 0
  >>= λv → 1  @:: editTask "Ready" 0
  >>= λw → 0  @:: editTask "Result" (v+w),
```

User 0 (a login procedure binds a user to a unique id) has to enter a number, then user 1 has to enter a second number, then user 0 gets the sum of the numbers, but

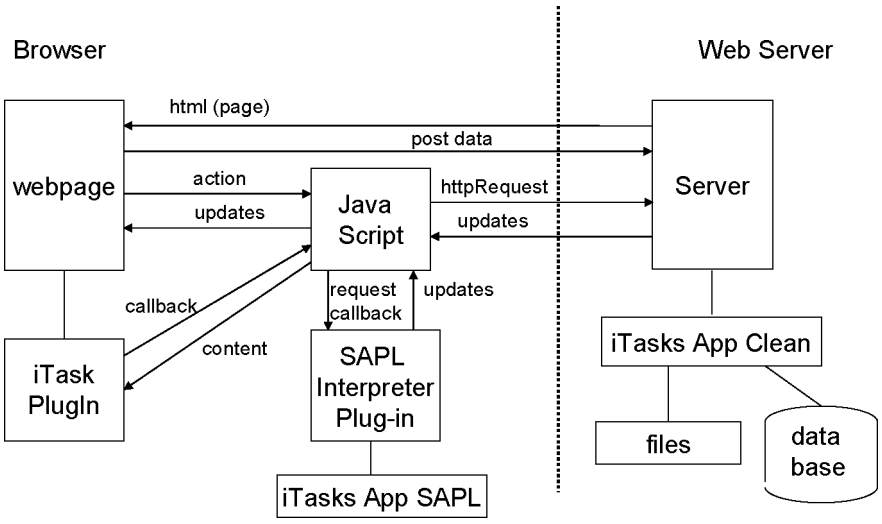


Figure 7.3: The architecture of iTask with plug-ins

can still edit the result.

The =>> operator is the iTask equivalent of the monadic 'bind' operator. With `n @:: task` one assigns `task` to user `n`.

7.3 iEditor: Plug-ins in iTask

Plug-ins are used for features that are not supported by standard HTML constructs like interactive drawing, complex text editing and animations. Plug-ins have to be installed by the user of the browser. Once this is done, they can be loaded by special HTML constructs. The use of plug-ins however, complicates the development of web applications. The developer has to take care that the plug-in is initialized and that the data needed by the plug-in is passed to it. In some cases, data from the plug-in has to be passed back to the web application or events occurring in the plug-in have to be handled by the web applications (e.g. mouse events).

In this section we introduce iEditor, an extension to iTask for the integration of plug-ins and give an example of its use.

7.3.1 The iTask architecture including iEditors

In Fig. 7.3 we show the adapted iTask architecture for including iEditors. The extensions with respect to the standard iTask architecture (Fig. 7.1) are:

- The plug-in is part of a web-page. This means that the initial web-page should contain an HTML representation of the plug-in;
- All communication with a plug-in must be done via JavaScript functions. This is the standard way of communication with plug-ins. All popular plug-ins can be accessed from JavaScript and can call JavaScript functions. Although it is

possible to communicate directly with Java Applets from the Sapl interpreter, we use the indirection via JavaScript to obtain a uniform interface that can also be used for non-Java plug-ins;

- For call-backs from the plug-in, the JavaScript function handling them has to decide where the calls should be made: either server or client.

7.3.2 The `PlugIn` wrapper type

In a basic iTask an element of a data type is turned into an editable HTML form by the function `editTask` and the result of editing the form is automatically turned into an updated instance of this data type. We want to maintain this interface for iEditors. More concretely, the information exchanged with a plug-in must also be represented by a data type and the use of the plug-in should lead to an updated instance of this data type. Because `editTask` has no means of distinguishing a data type intended for a plug-in from any other data type and also because we need information about how to load and display the plug-in, we have to wrap the content data type into a special `PlugIn` data type. For this wrapper type we can now make a specific implementation of the `editTask` function.

```
::PlugIn ct et st = {plugininfo    :: PlugInInfo,
                     content       :: ct,
                     events        :: [et],
                     state         :: st,
                     callback      :: [et] (ct,st) → (ct,st),
                     isServerEvent :: et → Bool}
```

The wrapper type contains all information needed for the creation of the plug-in (the right HTML code). It also contains all information needed to enable communication from plug-in to JavaScript and vice versa. `PlugIn` has three type parameters `ct`, `et` and `st`:

- `ct` is the type of the content to be exchanged with the plug-in;
- `et` is the type of the events that can occur in the plug-in;
- `st` is the type of the state that must be maintained between calls of the call-back. This type is not visible to the plug-in itself, but only to the call-back function that handles events from the plug-in.

The fields in the PlugIn type have the following meaning:

- `plugininfo`: information for constructing the HTML representation of the plug-in: how to load the plug-in, its size and other initializing parameters (see the example in Section 7.3.4);
- `content`: content of the plug-in. This field contains the initial content of the plug-in and after the plug-in is ready it contains the result of the plug-in;
- `events`: generated events that have to be processed by the call-back function;

- `state`: value of the state to be maintained between call-back calls;
- `callback`: call-back function that handles the generated events;
- `isServerEvent`: indication where events have to be handled.

The call-back function takes the generated events, the current content and state as input and returns a new content and state. The content is passed back to the plug-in. The state is maintained for the next call of the call-back. The call-back function is automatically called from the plug-in whenever an event occurs. On the plug-in side there should be data types to which the content and event types can be mapped (more details in Section 7.4). Mismatches will lead to the generation of exceptions on either Clean or plug-in side.

For indicating where events have to be handled, the user must specify the function `isServerEvent`. If this function returns `True` for an event, this event is handled on the server; if it returns `False`, the event is handled on the client.

From the iTask point of view an iEditor is just another editor for a data type (the content field). All other information in the `PlugIn` type is only there for enabling the creation of the iEditor and for doing processing (event handling) for the plug-in (invisible at the iTask level). For plug-ins not requiring event processing, the `events`, `state` and `callback` fields can be filled with stubs.

7.3.3 Interface functions for a Plug-in

For exchanging information between the iTask program and the plug-in two interface functions (one for the plug-in and one for JavaScript) are needed:

```
setContent(String content)
doPlugInCall(String pluginid, String content, String events)
```

`setContent` should be implemented by the plug-in and must be callable from JavaScript. `doPlugInCall` is a JavaScript function and must be called by the plug-in. `pluginid` is a unique id, identifying the plug-in (there can be more than one plug-in). The `content` and `events` arguments are serialized versions of the corresponding Clean datatypes (see Sections 7.3.4 and 7.4). For Java Applets we provide a generic Java class that takes care of the communication between plug-in and iTask program (see Section 7.5).

For other plug-ins there are two possibilities. Either the plug-in should be adapted by wrapping code that supports these functions, or special interface code can be written in JavaScript taking care of the conversion of Clean data to data compatible with that of the plug-in. Often, this interface code can be used for a whole class of similar plug-ins.

7.3.4 A Graphical Editor Plug-in for iTask

We now look at an example of the inclusion of an iEditor in iTask: a simple graphical editor. We assume, we have created a Java Applet plug-in that is capable of displaying simple graphics (lines, ovals, rectangles, etc.) and that can generate events

for mouse and button actions. The processing of events depends on what kind of graphical editor we want to make (vector graphics, diagrams, etc.). It is possible to create a dedicated plug-in for each kind of editor, but by using Clean for doing event handling we can adapt the behavior of the application by only changing the Clean source, without the need to adapt the plug-in. The key idea is that mouse and button events are passed to the web-application by a call-back function call. The call-back function can either be executed on the server by the Clean executable or on the client by the Sapl version of the Clean application. For each type of event, the programmer can choose where it must be handled.

We give the (almost) complete Clean source code for this editor. We start with the data types:

```
::GraphObject  = GraphLine Int Int Int Int | GraphOval Int Int Int Int |
                 GraphRect Int Int Int Int | GraphPolyLine [Pnt]       |
                 GraphButton String
::Pnt          = Pnt Int Int

::GraphEvent   = MouseDown Int Int | MouseDrag Int Int |
                 MouseUp   Int Int | ButtonEvent String

::GraphState   = NewLine |  NewPolyLine | NewRect | NewOval
```

In the application a drawing is represented by a list of `GraphObject`. We distinguish several types of figures and simple buttons (for the sake of simplicity we combined figures and buttons in one type). `GraphEvent` represents the events that can occur. We distinguish mouse (down, up, drag) and button events. The `Int`s represent the `x` and `y` position of the mouse event We assume that the plug-in is capable of displaying elements of `GraphObject` and that it turns events into elements of `GraphEvent`. The plug-in should have matching types for `GraphObject` and `GraphEvent`. The transformation of elements of these types onto each other is done automatically (see Section 7.4 and 7.5). `GraphState` is a state data type maintaining that part of the state that is not passed to the plug-in, but that is needed by the call-back function. In this example it maintains the type of the figure to be drawn at a mouse down event.

The task definition is given by:

```
graphtask ::  Task (PlugIn [GraphObject] GraphEvent GraphState)
graphtask =   editTask "Ready" graphplugin
```

The initialization of the plug-in is given by:

```
graphplugin :: PlugIn [GraphObject] GraphEvent GraphState
graphplugin =  {plugininfo    = grapheditapplet,
                content       = initpicture,
                events        = [] ,
                state         = NewLine,
                callback      = doEvents,
                isServerEvent = isMouseUp}
```

```
isMouseUp (MouseUp _ _) = True
isMouseUp           _   = False

grapheditapplet = AppletPlugIn {id      = "drawplugin",
                                archive = "drawapplet.jar",
                                code    = "drawapplet/maincanvas.class",
                                width   = 500,
                                height  = 200}

initpicture = [GraphButton "Line",      GraphButton "PolyLine",
               GraphButton "Rectangle", GraphButton "Oval"]
```

`graphplugin` contains the initialization of the plug-in. We see that all events are handled on the client except `MouseUp` events. As a consequence, the server side `PlugIn` data type is updated at every `MouseUp`. `grapheditapplet` contains the information needed for generating the HTML representation of the plug-in: the Applet id, the codebase and main class, its width and height. Finally, we see that the initial picture only contains the buttons.

Events occurring in the plug-in, are handled by the `doEvents` function:

```
doEvents ::     [GraphEvent] ([GraphObject], GraphState) →
                             ([GraphObject], GraphState)

doEvents [ButtonEvent "Line":evs]   (figs,_)
= doEvents                   evs    (figs,NewLine)

doEvents [MouseDown x y:evs]        (figs,NewLine)
= doEvents               evs        ([GraphLine x y x y:figs],NewLine)

doEvents [MouseDrag x y:evs]        ([GraphLine v w _ _: figs],a)
= doEvents               evs        ([GraphLine v w x y: figs],a)

doEvents [e:evs] (figs,a) = doEvents evs (figs,a) // ignore other events
doEvents []      (figs,a) =              (figs,a) // return result
```

Here only the code for `Line` is shown, `Rectangle`, `PolyLine` and `Oval` are handled in a similar way. Fig. 7.4 shows a screen shot of the application.

The user can stop editing by clicking the 'Ready' button. The current content and state are now made available to the remainder of the iTask application.

A multi-user graphical editor

To show that the plug-in task simply behaves like a normal iTask we give a small variation of `graphtask` analogous to the multi-user example from Section 7.3:

```
graphtask :: Task (PlugIn [GraphObject] GraphEvent GraphState)
graphtask =             0 @:: editTask "0 Ready" graphplugin
          ⋙= λv → 1 @:: editTask "1 Ready" v
          ⋙= λw → 0 @:: editTask "Result" w
```

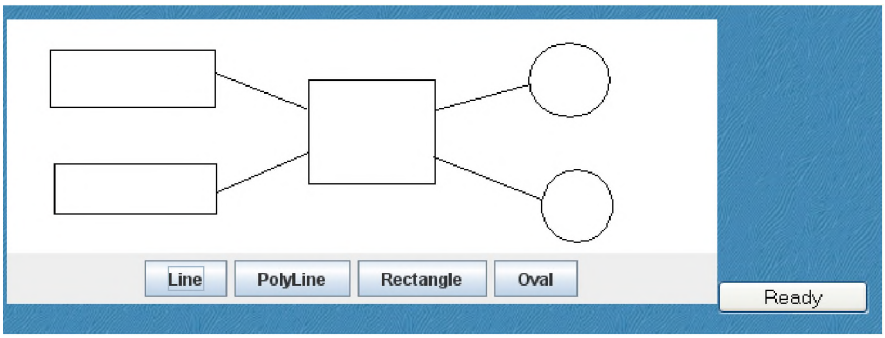


Figure 7.4: Screen shot of the drawing application

Two users are involved in this example. User `0` makes an initial drawing. The result is passed to user `1`, who can further edit the drawing. If this task is ready the result is passed back to user `0` who can continue editing.

For this application the programmer only has to specify the (content, state and event) types and the call-back function needed for handling events. The plug-in iTask behaves like an ordinary iTask. All communication with the plug-in is handled in a way that is transparent for the programmer.

It is also possible to wrap several plug-ins into one task. For example: `editTask "Ready" (graphplugin,texteditplugin)` wraps two editors together into one form. The editors are displayed next to each other.

7.4 Implementation of iEditors

From the example it is clear that the use of iEditors is straightforward for the application programmer and that, from the iTask point of view, an iEditor is just another editor. In the implementation we face a number of challenges (to answer questions 1 to 4 from Section 7.1):

- How to fit the plug-in into the iData/iTask architecture?
- How to exchange data between server, plug-in and client?
- How to invoke call-back functions from plug-in for server and client?

7.4.1 Fitting a plug-in into the iTask architecture

The HTML representation of the plug-in is generated as part of the initial iTask web-page. This is realized by making a specialized implementation of the generic `gForm` [PA06b] function that is part of the implementation of `editTask` and that is responsible for the generation of the web form. The resulting HTML also contains the initial content of the plug-in and all other information needed by the plug-in and by the JavaScript functions that interact with the plug-in. This adapted `gForm` is generic and works for all plug-ins.

7.4.2 Data exchange between client, plug-in and server

For data exchange between plug-in, server and client Clean program we use the generic print and parse functions on the Clean side (on server and client). As a consequence the plug-in must have a (generic) way to parse and unparse the strings representing the event and content data types. In Section 7.5 we discuss how this is realized for Java.

Although we have the same Clean program running at both the server and client side, the internal representations of data types are completely different. Therefore we also use the generic print and parse functions for the exchange of data between the Clean programs at server and client side.

All communication between server, client and plug-in is done using JavaScript functions, similar to what is done for Ajax and client-side handling of iTask tasks (see [PJKA08] or chapter 6)). These functions are generic in the sense that they do not depend on the specific plug-in. The JavaScript functions are responsible for passing data from plug-in to server and client and vice-versa, but also for making the call-back and deciding where the call-back must be handled. The addition of plug-ins only requires one extra JavaScript function to handle all communication from the plug-in with client and server Clean programs:

```
doPlugInCall(String pluginid, String content, String events)
```

This function can both handle the final result from a plug-in and the call-backs generated by the plug-in. The first argument is the unique `ID` of the plug-in (there can be more than one plug-in and they all use this JavaScript function). The second argument is the serialized version of the current content of the plug-in. The third argument is a serialized version of the list of the events that must be processed (this list is empty in case the plug-in just wants to synchronize its content with the server program). The JavaScript function takes care of either updating the server program with the content of the plug-in or by making the call-back to client or server program (see Section 7.4.3).

For updating its content the plug-in should implement the following function:

```
setContent(String content)
```

The argument is again a (serialized) string representation of the content. This function is called from JavaScript. It is custom for plug-ins to support function calls from JavaScript.

7.4.3 Handling call-backs

Call-backs can be made to either client or server. The plug-in makes the call-back by calling the (generic) Javascript function `doPlugInCall` with the serialized content and events as arguments. The JavaScript function determines whether the call must be handled on the server or the client by executing the `isServerEvent` function for the event in the Sapl interpreter. For the server case, the `PlugIn` data type is updated in a similar way as for an ordinary update for iData [PA06b]. The implementation of `editTask` makes use of the generic function `gUpd` for updating the data type with

the result of a user edit action. For the `PlugIn` data type a specialized version of `gUpd` is made that applies the call-back function to its arguments before updating the `PlugIn` data structure with the result. Finally, the HTML representation of the plug-in is re-generated with the new content.

We could use the same strategy for the client side (the full Clean program is available). But we must do it in a much more efficient way, because the overhead of finding out for which task the update is intended can be large. We can do it more efficiently because we have an interpreter available that can execute an arbitrary Clean expression. We use this to directly execute the call-back function call and use the result to make a direct update of the content of the plug-in. In this way we short-circuit the use of `gUpd` and the whole iTask machinery needed for finding out which task is updated [PAK07]. This optimization is absolutely necessary for events needing immediate response like mouse drag events.

For making the direct call-back on the client we use Clean-SAPL dynamics (see Section 7.2.2). For this, the serialized call-back function is stored in the plug-in HTML representation. Not only is the call-back function itself serialized, but the `isServerEvent` function and the parse and unparse functions for the arguments (content, state and events) are also serialized. The last is necessary because the arguments are passed to the call-back as serialized strings from the plug-in via the `doPlugInCall` function and the result must be passed back in serialized form too.

In the actual call-back, it is first checked if the event is really intended for the client by applying the `isServerEvent` function to the deserialized event. If not, the server call-back is made as described above. Otherwise the call-back and parse and unparse functions are all de-serialized, the arguments are parsed, the call-back is applied, and the result state and content are unparsed again. The content is handed back to the plug-in directly (via `setContent`) and the state is maintained in the HTML representation of the plug-in.

Note that we cannot handle all call-backs at the client side. Processing intensive call-backs and call-backs requiring information from data bases or files should be handled on the server side.

7.4.4 Evaluation of Efficiency of handling call-backs

In the graphical editor application we used the call-back function to handle mouse down and drag events by the SapI interpreter. Mouse drag events often occur in quick sequences (in the order of 10-15 events per second). The whole call-back machinery was capable of keeping track of these events on an Intel 1.6 GHz Core Duo 2 machine (using only one core). Attempts to handle the drag events by the server led to a browser hang-up due to a client-server communication overload. Of course, the (de)serialization of data types takes a significant amount of time and is a limiting factor in the amount of events that can be handled. Native implementations (without the need to (de)serialize) can easily handle up to ten times as much events.

7.5 Implementation for Java Applets

Java Applets [GJS96, Sun08] are an important class of plug-ins. All modern web browsers offer the possibility of Applet plug-ins. In this way it is possible to incorporate complex Java applications into web pages. We already used the Java Applet mechanism for loading the Sapl interpreter at the client side of iTask for handling client tasks and call-backs. Although Java Applets can offer rich functionality they are less popular, because communicating with them must be handled in an ad-hoc manner, making it difficult to integrate them with the remainder of a web application (see also [Rei98]). By using the iTask plug-in techniques, we have a generic strategy which simplifies the communication with Java Applets. To include a Java Applet in an iTask application we have to deal with the following issues:

- We have to find a way to map Clean types to corresponding Java data types;
- We have to take care that we offer the interface needed for communication with JavaScript.

7.5.1 Mapping Clean and Java Data Types onto each other

In order to exchange information between a Clean and Java application there must be a way to transfer Clean data to Java data and back. To save the programmer from writing boilerplate data transformation code we included generic code in Clean and Java to handle this data transformation. Not all Clean and Java data types can be mapped onto each other. For a Java class the member fields are (currently) restricted to the following types:

- primitive types: (`int`,`long`,`float`,`double`,`boolean`,`char`);
- the `String` type;
- all subtypes of `List` (they are all mapped on a Clean list);
- other Java classes with members that obey these rules.

A class may be a subclass of another class or implement an interface, but all superclasses must obey the rules mentioned above. Other (container) types, like `Map` and arrays are not (yet) allowed. For these classes an ad-hoc mapping must be made (like is done for `List`).

From the Clean point of view, the automatic conversion of Clean data types to Java types is restricted to first order data types that can be described by standard Algebraic Data Types. Records are not allowed yet, but they can be easily added. If a Clean type is mapped onto a Java type hierarchy the fields of the Clean type should match the union of all fields in the class hierarchy in the order of the hierarchy (fields of superclass before fields of subclass).

More formally, consider the following algebraic data type definition in Clean:

```
::typename t1 .. tk = C1 t11 .. t1n_1 | .. | Cm tm1 .. tmn_m
```

`t1..tk` are type parameters, `C1..Cm` constructor names and `tik` type names or type parameters. This type definition corresponds to the following Java interface and `m` Java classes:

```
interface typename {}
class<t1,..,tk> C1 implements typename {t11 a11; .. t1n_1 a1n_1;}
...
class<t1,..,tk> Cm implements typename {tm1 am1; .. tmn_m amn_m;}
```

Each constructor is represented by a separate Java class with as name the constructor name and with as fields the arguments of the constructor (with names `aik`) with their type. As an example, the Clean type `GraphObject` from Section 7.3 corresponds to the following Java classes:

```
interface GraphObject {}
class GraphLine implements GraphObject {int x, y, v, w;}
class GraphRect implements GraphObject {int x, y, v, w;}
class GraphOval implements GraphObject {int x, y, v, w;}
class GraphPolyLine implements GraphObject {List<Pnt> points;}
class GraphButton implements GraphObject {String name;}

class Pnt {int x, y;}
```

It is possible to generate a corresponding Clean data type definition from an existing Java class (hierarchy) using generic Java functions. Otherwise the programmer has to take care that matching types at Clean and Java side exist, as we did in our graphics editor example. Once corresponding data types exist, the conversion of data is done automatically.

For the actual conversion of data we use the standard generic print and parse functions at the Clean side (`gPrint` and `gParse`) and reflection [McC98] on the Java side.

7.5.2 Java Applet Plug-In Interface

To further simplify the communication between Java plug-in and Clean iTask application, the Java class `CleanJavaCom` is offered. This class contains member functions that can be used for parsing and unparsing Java objects and functions for handling the communication with the JavaScript interface. The `CleanJavaCom` class is generic and can be used in every Java Applet to be used as plug-in in an iTask application.

```
class CleanJavaCom<CT,ET> {
  private String writeClassToString(Object object)  {...}
  private Object readClassFromString(String inp)    {...}
  public CT getContent()                            {...}
  public void setContent(String ser_content)        {...}
  public void handleEvents(List<ET> events)         {...}
  ...
}
```

The class is parametrised by the Java versions of the content (`CT`) and event (`ET`) types.

- `writeClassToString` generates a string representation of an object that exactly fits the Clean representation;
- `readClassFromString` parses a string representation generated by `gPrint` to the corresponding Java object;
- `getContent` is called to obtain the content by the remainder of the Applet;
- `setContent` is called from JavaScript to set the content. The content string is de-serialized by a call to `readClassFromString`. The result can now be obtained by the remainder of the applet by a call of `getContent`;
- `handleEvents` is called by the Applet after one (or more) event(s) have occurred. This function takes care of serializing the content and events and calling the `doPlugInCall` function in JavaScript. JavaScript should pass back the result by a call of `setContent`. If the plug-in only wants to synchronize its content with the iTask server application it should call `doEvents` with an empty event list.

In the implementation of `writeClassToString` and `readClassFromString` the Java reflection mechanism is used.

7.6 Discussion

Plug-ins must have matching types for the content and event types. For Java we implemented a generic way to convert the serialized content and event types to Java data structures and back. Not all plug-in types offer the possibility to do this conversion in a generic way. An alternative is to use generic functions in Clean for generating a representation the plug-in can deal with and for parsing back the results. An example is the use of XML [BPSM98]. Java has the `XMLEncoder` and `XMLDecoder` classes for generating and parsing XML representations of data types. For us, a more interesting alternative is the use of JSON (JavaScript Object Notation) [JSO]. This has as an advantage that we can also exchange data with JavaScript and a large number of other formalisms. Like string serialization, it allows for a lightweight implementation with little overhead. We already started to implement generic generation and parsing of JSON data in Clean and we will use this for future implementations.

Other alternatives are the use of CORBA [OMG96] or the Java Native Interface [Lia99] for exchanging data between Clean and the plug-in. Examples can be found in [MF00, Rei98, ER97]. For us, these approaches are too heavyweight to be used at the client side in the Sapl interpreter.

The idea of attaching an event handler to an editor is not restricted to plug-in tasks. Call-back functions can also be attached to other basic iTask editors. The call-back can be used to check the content of the editor before sending it back to the server and give the user feedback in case something is wrong or to reformat the content before displaying it again. Because the full power of Clean is available at the

client side there are no restrictions to the call-back functions that can be defined. In this way, the concept of iEditors can be extended to arbitrary iTask editors. To implement this, we can either use a wrapper type like `PlugIn` or introduce a special combinator in iTask like the one used for assigning users to tasks (see Section 7.2).

7.7 Related Work

In this paper we extended the iTask toolkit with a generic framework for the inclusion of plug-ins, with the possibility to make calls from the plug-in to Clean functions that can be executed on either client or server. We are not aware of any other functional system that has these features. However, there are functional approaches for handling web pages using the same formalism for server and client-side processing. Most of them compile to JavaScript for client-side execution. An example of this approach is Hop [SGL06, LS07]. Hop is a dedicated web programming language and its syntax is HTML-like. In Hop it is also possible to specify a complete web application without the (direct) use of JavaScript. Hop uses two compilers, one for compiling the server-side program and one for compiling the client-side part. The client-side part is only used for executing the user interface. The application essentially runs on the client and may call services on the server. Hop uses syntactic constructions for indicating client and server part code. It is build on top of the Scheme programming language. In our case we do not have to extend Clean, but can write the entire web application in Clean itself. In [LS07] it is shown that a reasonably good performance for the client-side functions in Hop can be obtained. For us, compiling to JavaScript is no option because Clean is lazy. Instead we use the SapI interpreter, which also has competitive performance as was shown in [JKP06] (chapter 3) and the graphics editor application.

Links [CLWY06] and its extension formlets is a functional language-based web programming language. Links compiles to JavaScript for rendering HTML pages, and SQL to communicate with a back-end database. A Links program stores its session state at the client side. In a Links program, the keywords `client` and `server` force a top-level function to be executed at the client or server respectively. In Links, processes can be spawned and these processes can communicate via message passing. Client-server communication is implemented using Ajax technology, like we do. In the iData and iTask toolkits, forms are generated generically for every data type, whereas in Links and Formlets these need to be coded by the programmer.

The Flapjax language [Kri07] is an implementation of functional reactive programming in JavaScript, with features comparable to those of Hop. Both are designed to create intricate web applications. In Flapjax, Hop and Formlets processing is directly attached to web form handling, which is comparable to the use of call-backs in iEditors.

A much more restricted approach has been implemented in Curry [Han07]: only a very restricted subset of Curry is translated to JavaScript to handle client-side verification code fragments only.

Summarizing the main differences with the other approaches are:

- iTask/iEditor applications are just plain Clean applications, where web forms are generated from data types. The other approaches define dedicated web languages where processing is attached to web forms;

- We can use the full Clean functionality at the client side because the Sapl interpreter offers a full Clean platform. The other approaches rely on compilation to JavaScript with, in many cases, restrictions on the functions that can be compiled to JavaScript;

- Clean-SAPL dynamics offers a generic and flexible way to attach call-back handling to web forms and plug-ins. Where the other approaches use static annotations to indicate whether functions have to be executed on either client or server, in our approach this can be decided dynamically, depending on the events to be processed.

7.8 Conclusions

Plug-ins are often an essential part of more interactive web applications. In this paper we discussed a generic way for including plug-ins in iTask applications. All communication between iTask application and plug-in is on the level of exchanging and updating data types, which is entirely consistent with the normal way iTask works. Plug-in tasks behave like ordinary tasks. No adaptations of iTask were necessary to incorporate them, only a specialization of the `gForm` and `gUpd` functions for the PlugIn type.

An important feature is that plug-ins can use Clean functions, which can be executed on either server or client, for event handling. This gives the programmer fine-grained control over the behavior of the plug-in without the need to adapt the plug-in itself. In this way, we can keep the plug-in to its essence and use Clean for all processing not involving the specialities of the plug-in.

Information exchange between server, client and plug-in is realized with the use of generic (un)parsing of data types. For efficient client-side event handling a combination of Clean-SAPL dynamics and generic (un)parsing is used. With Clean-SAPL dynamics it is possible to move the execution of arbitrary Clean expressions from server to client. This turns out to be a powerful feature that can also be used for attaching client-side functions to arbitrary web forms.

For Java Applets, a straightforward to use generic class is provided that handles all interaction of the plug-in with Clean including the conversion of data types and the forwarding of call-backs. Plug-ins of other type should implement a simple JavaScript interface and the (de)serialization of the data types used for the exchange of information.

We have maintained the declarative approach of the iTask toolkit. Server and client programs and all call-back handling functions are generated from an annotated, single-source specification with a low burden on the programmer because the system itself switches automatically between client and server-side evaluation of

tasks and call-backs when this is necessary. The iTask system integrates all mentioned technologies in a truly transparent and declarative way.

Chapter 8

Web Based Dynamic Workflow Systems for C2 of Military Operations

[1] ***Abstract*** Modern military operations are complex endeavours involving different services and nations, as well as governmental, commercial, non-governmental, and international organizations. Each partner may have its own planning process and tools. This diversity must be orchestrated to plan the overall operation, while maintaining the agility to respond to changing situations.

We contend that dynamic workflow mechanisms are suited to planning military operations. We developed the iTask dynamic workflow system that enables the construction of high level multi-user workflow applications. Workflows can change in response to dynamic situations, new tasks can be spawned by or be dependent on previous tasks, tasks can be dynamically adapted. iTask based applications have a web-based user interface, allowing external partners to use them without installing special software. Moreover, new parties can join them on-the-fly. Because of its focus on dynamic processes iTask appears promising for development of flexible C2 systems.

In this paper we present a discussion of the potential of the iTask system for building C2 systems. We give an overview of the system, and discuss to what extent it meets the requirements of the C2 domain. We also sketch a number of promising application areas in this domain.

8.1 Introduction

In modern warfare many activities have to be deployed by many people using a great diversity of systems. The coordination and control of these activities is becoming more and more complex. The advent of NCW and NEC [AGS99] complicates this even more, because much more information is becoming available in an even shorter time frame. Decisions are not taken in centralized headquarters anymore, but are the result of a collaborative effort of many. Command and Control has extended from a single system/group activity to a networked activity involving many systems

[1] Submitted as [JLPG10] and extended version of [JLP10] and [JKP08b]

and people distributed over large areas. Often non-military like local authorities and non governmental organizations (NGO's) are involved in operations. Also the nature of military operations has changed. Asymmetrical operations have become the standard. Information is the most important weapon in these operations. But obtaining information is difficult and requires other sources then the traditional sensors. Instead complex intelligence operations are required. Through these developments the borderline between military operations and response operations for crises, whether caused by aggression or caused by accidents or natural disasters, is fading. Systems that take care that information is made available to the right persons at the right moment, and that support the gathering of information become ever more crucial for successful execution of operations.

In recent years, the focus for research has been on the development of systems to enhance the quick sharing of information using Web 2.0 technology and Service Oriented Architectures (SOA, [IEH09]). Although making information available to all parties is crucial to enhance situation awareness [EBJ06], a recurrent theme in our discussions with military and crisis-management professionals has been that the real challenge is coordination and control. At first glance, Workflow Management Systems appear to have potential to support this (see also [SB09], [PLZ09] and [FW08]). WFMS's are computer applications that coordinate, generate, and monitor tasks to be performed by human workers and computers. Every activity in an operation can be considered a task. Activities can depend on each other and must be performed in sequence, while other activities may be carried out in parallel. The workflow system can be used to support the distribution and monitoring of these activities. But there are some serious problems, as already acknowledged by [SB09, PLZ09, FW08]. First, contemporary workflow systems are commonly rather rigid because they only model the static flow of control. Second, in many cases the activities to be conducted for executing an operation cannot be captured in a predefined plan. Only a rough sketch of the actions to be taken can be given. Plans can be further refined only at runtime, when more information becomes available. Most workflow systems cannot deal with this. They only offer the execution of detailed predefined plans. In other words, contemporary workflow systems are not capable of dealing with the dynamic nature of modern military and crisis response operations where tasks may heavily depend on the outcome of previous tasks and plans must be changed on-the-fly due to changing circumstances.

Recent work on the use of functional programming techniques for workflow modeling has led to the development of the iTask system [PAK07, PAK08a]. The iTask system is a domain specific workflow language embedded in the functional programming language Clean, enabling the creation of data-driven dynamic workflow systems. It supports data dependent behavior of tasks, where the new tasks to do may depend on the results of previous tasks. The iTask system allows for on-the-fly (dynamic) adaptation of tasks. A full (extended prototype) implementation of the iTask system is already available. A number of iTask applications have been implemented including a conference management system (see [PAK+08b]) and a number of smaller example applications. It should be noted that iTask is not a C2 system itself, but a toolkit for the construction of such systems.

In this paper, we present a discussion on the suitability of dynamic workflow specification, and its implementation using the iTask system, for modern C2 of both military and crisis response operations. We also sketch a number of candidate application areas for iTask and indicate what the possible gains are for these areas. We use five key design requirements for response technology proposed by Jul in [Jul07] as a framework for structuring our discussion. The current iTask system already offers important functionality for supporting C2 operations, and can be trivially extended to meet even more. However, we also identified a number of research challenges that need to be addressed to fully optimize the iTask system for supporting C2. Although some of the strengths, weaknesses and challenges we discuss apply only to the iTask system, most apply to workflow management systems in general.

8.2 The iTask system

The iTask system (itasks.cs.ru.nl) is a domain specific workflow language embedded in the functional programming language Clean, enabling the creation of dynamic data dependent workflow systems. This means that it enables programming of workflow systems in a programming language that is specifically tailored for this purpose, but at the same time has the full computational expressiveness of a modern functional language. A workflow system is data dependent if it allows for adaptation of workflows using intermediate results. In the iTask system a workflow consists of a combination of **tasks** to be performed by humans and/or automated processes. From iTask specifications complete workflow applications are generated that run on the web, optionally distributed over servers and clients [PJKA08] (chapter 6). The iTask system is based on open web-standards and can therefore be accessed by anyone who has access to Internet, nowadays including many mobile devices. The system has an interface resembling e-mail client software to reduce the need for users to learn additional interactions.

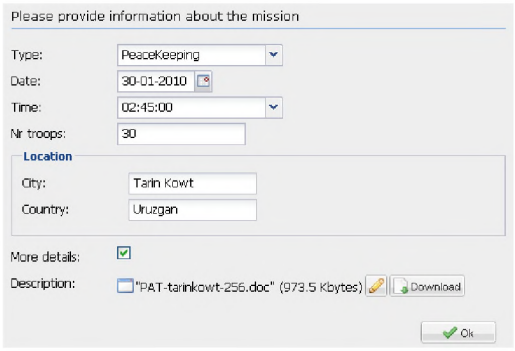
The iTask system is built upon a few simple concepts. The main concept is that of a typed task. A task is a unit of work to be performed by a worker or computer (or a combination of both) that produces a result of a certain *type*. Result types are not limited to simple data such as integers, records, etc., but can also be documents, or even new tasks. The result of one task can be used as the input for subsequent tasks, and therefore these new tasks are dynamically dependent on this result. This also holds for tasks that produce or consume other tasks.

We distinguish two kinds of tasks: basic tasks and composed tasks. Basic tasks are elementary tasks that can be fulfilled by one user in one step. In the workflow language these are black box primitives that are implemented at a lower level. An example of a basic task would be the entering of information in a web-form by a user. The result of this task is then the entered data. Composed tasks, or workflows, are defined by composition of (basic) tasks using so-called *task combinators*.

8.2.1 Basic iTasks

The iTask standard library offers several functions for creating basic units of work: basic tasks. An important example is the *generic* task where a user is asked to supply information. The generation of a web-form to enter this information and the processing of its result are handled fully automatically by the iTask system. In this way one can create data-entry tasks in just a single line of code. Figure 8.1 shows the code for a task where the user can supply the information for a military mission, together with a picture of the generated form. The code consists of data type definitions and a task definition (`enterMission`) using these data types. This example also shows that documents can be attached to tasks.

```
:: Mission =
    { type        :: MissionType
    , date        :: Date
    , time        :: Time
    , nrTroops    :: Int
    , location    :: Location
    , moreDetails :: Bool
    , description :: Document
    }
```




```
:: MissionType = PeaceKeeping | CounterTerrorism | SpecialService | IntelOperation |
                 War | Other String

::Location = {city::String,country::String}

enterMission :: Task Mission
enterMission = enterInformation "Please provide information about the mission"
```

Figure 8.1: A Generic Data-Entry Task for `Mission` Data.

An obvious advantage of such compact definition of data-entry tasks, is that it enables readable and easily modifiable workflow specifications. But there are some less obvious, but more important ones: First, the separation of declarative task definition and generic implementation enables different implementations for different devices. Second, because interfaces can be automatically generated, the system can automatically provide a fallback based on manul data entry for *every* task. Even for tasks that were designed to receive their input through an automated process.

Other examples of basic task functions are: requesting lists of users of the system (if necessary grouped by their role); tasks that return at a predefined moment in time or after an amount of time; tasks that communicate with other applications, databases, sensors, or web services (for the exchange of information). We have for example implemented access to Google Maps.

8.2.2 iTask Combinators

New tasks can be composed from other tasks by using *combinator* functions. We distinguish between combinators that say something about the order in which tasks have to be performed and combinators that say something about an individual task: who has to perform it; where to store information about the task, etc.

Tasks can be organized in many ways. In most contemporary workflow systems organization is limited to a fixed set of patterns (see [AHKB02]). Because the iTask language is embedded in an expressive general purpose host language (Clean) all common patterns, and many new patterns can be expressed using using relatively few combinators. Here we discuss the most important ones.

Sequential Combination

In contrast to most workflow specification languages, information is passed explicitly from one task to another in the iTask formalism. In a sequential composition of two tasks, the first task is activated first and when it finishes, the result is passed to a second task, which takes this result as its input. In code this is denoted by:

```
first_task >>= second_task_function
```

Note that t `>>=` f (or t followed by f) integrates *computation* and *sequential ordering* in a single pattern. In this way the second task can dynamically adapt to the result of the first task. In other (mostly graphical) workflow formalisms it is harder to specify a function that acts on the result of a preceding task because only control is passed between tasks.

Parallel Combination

An important combinator for executing a number of tasks in parallel is the `parallel` combinator. Where other workflow formalisms contain a large number of patterns (see [AHKB02]) for executing tasks in parallel, iTask needs only one combinator for this. Using the power of the functional host language, one can construct all other patterns (and more) using this single combinator. This is hard to do in other workflow languages because these lack the right abstraction mechanism for realizing this. With the parallel combinator one can start the execution of several tasks in parallel and stop this execution as soon as a user specified condition is fulfilled. For example, one can stop when one task (or-parallelism) is finished:

```
anyTask [task_1,task_2,task_3,task_n]
```

When all tasks (and-parallelism) are finished:

```
allTasks [task_1,task_2,task_3,task_n]
```

Or when the results of the finished tasks satisfy a certain condition (ad-hoc parallelism):

```
conditionTasks condition [task_1,task_2,task_3,task_n]
```

These different combinators are all shorthands for the same generic `parallel` combinator instantiated with different parameters.

Task Assignment

Tasks can be explicitly assigned to users using the task assignment (`@:`) combinator.

```
userid @: task
```

Here the task `task` is assigned to the user with login name `userid`. The user to whom a task is assigned, can be entered explicitly in the workflow model. Alternatively, it is possible that during the execution of the workflow, a task determines the user to which another task must be assigned. Once tasks have been initially assigned, it is always possible to reassign them to another user on-the-fly. It is possible to monitor the progress of tasks. This information can be used to re-allocate tasks to different users, to stop tasks or to replace tasks by other tasks.

8.2.3 An Example iTask Workflow

A typical example of a task for which a dynamic workflow system can be used is the dynamic allocation of resources. Consider, for example, the following scenario. For a complex military mission transportation of people, equipment and supplies is necessary. The amount and kind of transportation devices heavily depends on the location of the mission area, the number of people and goods to be transported, the condition and safety of the transportation routes.

We implemented a prototype application that automates this process using the iTask system.

The starting point for this workflow is a mission report like the one described above. This report contains the type and the location of the mission. The workflow uses this information and a set of available transportation providers and their locations to calculate an initial set of requests to be sent to transportation providers to obtain the right amount of vehicles. Each provider should reply within a certain time limit whether it is capable of supplying the requested amount. In case not enough vehicles can be supplied the system automatically starts requesting other providers and recursively continues doing this until enough transportation capacity is available.

Due to space limitations, we only show the code of the first part of the workflow specification. This part handles an incoming report for a mission and starts the operation. It consists of three steps: First some data describing the mission is entered (`enterMission`), then the appropriate actions are chosen (`planActions`, and finally the actions set in motion (`allTasks`). During the second step, a suggestion for further action is computed based on the type of mission that is entered in the first step. During the third step all tasks that have been chosen as actions are carried out in parallel. Transportation is handled in the `orderTransport` task. The iTask code for this workflow is the following:

```
startMission
  = enterMission >>= planActions >>= allTasks
where
  enterMission :: Task Mission
  enterMission = enterInformation "Please provide information about the mission"
```

```
planActions :: Mission → Task [Task Void]
planActions mission
  = updateMultipleChoice "Choose actions" options (suggestion mission.type) ++
    if mission.moreDetails [detailSpecification] []

where
  //Generate the list of possible tasks to choose from
  options = [f mission \\ f ← [makeComsPlan,orderTransport,orderSupplies,orderAirSupport]]

  //Compute the indexes in the options list that are initially selected
  suggestion PeaceKeeping      = [0,1,2]
  suggestion CounterTerrorism  = [0,1,2,3]
  suggestion SpecialService    = [0,1]
  suggestion IntelOperation    = [0]
  .........
  suggestion _                 = []
```

This small piece of code already demonstrates two core features of the language. First, it integrates computation in the workflow. The `suggestion` function computes the initial selection of actions from the information that is entered in the `enterMission` step. In this case it is a simple one-to-one mapping of mission kinds to selections, but it is possible to do any computation to select or parametrize the next steps in a workflow. The second interesting feature is that the result type of the `planActions` task is a list of tasks. This list of tasks, which are all parametrized with the mission information, is then executed in parallel by the `allTasks` combinator. The possibility to have tasks that have new tasks as their result can be used to create highly dynamic models that contain steps in which parts of the worklow are interactively defined during execution. In `planActions` we also inspect the `moreDetails` field in the original form to decide whether or not a `detailSpecification` task should be started parallel to the other tasks.

8.2.4 Dynamic Behavior: Exceptions and Change

Several authors, like [SB09], [PLZ09] and [FW08], have already indicated that workflows need to be adaptive to be of use for complex domains like command and control and crisis-management operations. The iTask system offers a number of programming constructs to support the following kinds of dynamic behavior:

1. Dynamic behavior that can be anticipated and where the normal course of actions is not affected. In these cases the procedure to be followed depends on the intermediate results of previous tasks. This is considered as normal dynamic behavior and is provided by the sequence (>>=) combinator.

2. Dynamic behavior that can be anticipated where the normal course of actions is affected. In these cases normal procedures should be stopped and a different procedure should be started. The *exception* mechanism in the iTask system provides this capability.

3. Dynamic behavior that cannot be anticipated and detected within the workflow itself. In this case ad-hoc changes should be made to one or several workflows. The normal procedure should be stopped, and then either a different procedure should be started or an adaptation to a (sub)task should be made. This form of dynamics is provided by the *change* concept in iTask.

iTask supports an exception mechanism similar to what is found in common programming languages. A task may throw an exception in case an exceptional situation occurs. The entire workflow the task is part of is now stopped (if there are parallel tasks in it, the users participating in these tasks are informed). The exception is passed to an exception handler. This handler can now start a new task using information raised in the exception as its input. An exception must be explicitly thrown in a workflow. So, the designer of the workflow must be aware that exceptional situations may occur during the execution of a workflow and therefore has to define a handler for them. Exceptions enable the separation of uncommon borderline cases from the regular workflow.

The *change* concept is complementary to that of the exception. While an exception is the result of an abnormality that occurs within a workflow, a change is triggered from outside the specified workflow. Tasks on which people are working can be replaced on-the-fly with other tasks. Because iTask workflows are typed, the new task should return a result of the same type as the replaced task. An example of a change is the replacement of a complex process by an ad-hoc made to-do list, in case the user has determined that the process is inappropriate for the current situation. Another example is the replacement of a process by the ad-hoc entering of a result that is obtained in another way (not using the workflow system).

8.2.5 The iTask Client

In the iTask system, workflow instances are executed by a server application that is generated from the workflow specifications. These server applications disseminate information about tasks through a collection of web services.

Because end-users cannot access such services directly, the iTask toolkit provides a generic client application to let people view, and work on tasks. This client application, shown in Figure 8.2, is an Ajax application which is similar to a web-based e-mail client. But instead of an inbox of messages there is an inbox with interactive tasks. Because web-based e-mail applications are very common, this design requires a minimal amount of addtional learning.

The names of the tasks that the worker needs to perform are presented in the *task list* displayed in the upper right pane. This pane can be compared with the list of incoming e-mails. When the worker clicks on a task in the task list, its current state is displayed in the lower right *task pane*. Tasks can be selected from the task list in any order, allowing a local operator to determine a preferred order of execution. The iTask toolkit automatically keeps track of all progress, even if the user quits the system. When a task is finished, it is removed from the task list. Workers can start new workflows, by selecting them in the left *workflow pane*. In general any number

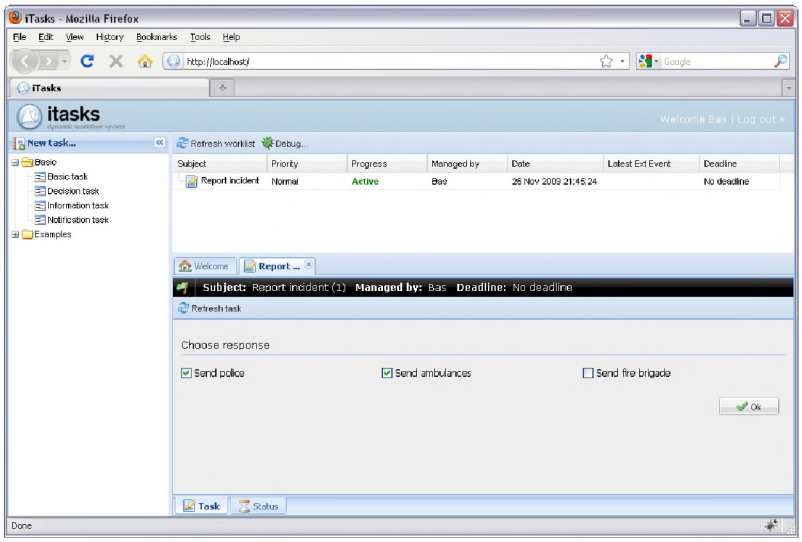


Figure 8.2: A Screenshot of the iTask client application

of workflows can be started. The task list is updated when new tasks are generated, either on the users own initiative, or because they have been delegated to it. The entire interface is generated automatically from a workflow specification.

8.3 Example Applications in the Military Domain

The introduction to the iTask system in the previous section, is necessarily dense and abstract due to the meta-system nature of the toolkit. Therefore, before continuing with a discussion of its suitability for military / crisis response operations based on requirements from literature, we first give some envisioned example applications areas in this section. Contrary to the iTask system itself, which is an implemented proof-of-concept system (previously published in [PAK07, PAK08a]), these applications are just examples to sketch a more concrete vision of its use in a military context.

8.3.1 Useful Characteristics

The iTask system can be used to make applications that tell people what to do at what moment. In this way iTask does not differ from other workflow systems and planning tools. More interesting are the unique properties of iTask that cannot be found in other workflow management systems.

First, workflows application programmed in iTask's are flexible in many ways. Because iTask allows for data dependent workflows intermediate results can be used to parametrize future steps. For example, if for transportation more vehicles are needed than can be supplied by the standard transport service, an additional workflow can be started for obtaining more transport capacity. Using iTask's dynamic data dependencies many dynamic aspects of operations can be captured. But not all

operations can be captured in predefined plans, because the steps needed to be taken during the operation cannot be determined exactly beforehand. But even in those cases the generic structure of the process is mostly known and only in the more detailed sub-procedures ad-hoc actions involving human improvisation are needed. For this we can use the *Change* concept of iTask. Using this *Change* concept workflows can be adapted in many ways. The most direct way to use it, is the replacement of (part of) a workflow by ad-hoc entering of information. This is necessary when someone decides that this part of the workflow is not appropriate for the current situation. Instead, the information needed at this point of the workflow is obtained in another (ad-hoc) way. This seems to be a trivial issue, but one often has 'to fight the system' because a procedure is not appropriate for the current situation, but one has to continue because there is no way to circumvent the system. The second way to use the *Change* concept is to do the exact opposite. In this case it is not possible at design time to give an appropriate workflow for a subtask at a certain point in the workflow definition. In this case a default workflow that just consists of a form where the user has to enter the appropriate information is offered in the workflow definition. During the execution of the workflow the user can decide to start a dedicated workflow that supplies the information needed at that point. The information that is generated by this workflow is now automatically entered into the other workflow.

Second, the exception mechanism can be used to stop an already running workflow automatically and replace it with another workflow. For example, a military patrol must be aborted, because new information shows that it is too dangerous to continue. Instead, an air strike action is necessary to clear the area. Stopping a workflow can also be done by a user with the appropriate rights. In case (part of) a workflow is stopped, the users involved in it are automatically notified and the results already obtained are discarded.

Third, it is possible for a user to construct ad-hoc workflows by making a composition of already existing workflows. The user is offered a dedicated interface to make simple sequential and parallel compositions of existing workflows.

Fourth, iTask is not a closed system and allows for easy integration of other web-services. This offers a straightforward way to extend the functionality of an iTask application and to use iTask as a web services coordination tool.

8.3.2 Preparation of Deployment for Military and Peace Keeping Operations

Preparations for military operations like the deployment of troops for peace keeping operations are characterized by their complexity and unpredictability. They are complex because large amounts of people and equipment have to be transported to very remote locations and every deployment has its own unique characteristics. They are unpredictable because it is almost impossible to use ready made plans to execute them and existing plans often have to be adapted due to unforeseen circumstances. The planning and execution of deployments comprises the following aspects:

Logistics Before people can be deployed, accommodation, power, water and food supply, etc. have to be arranged.

Transport Transportation is needed both for people and material (accommodation and supplies). A large part of the transportation has to be done beforehand (accommodation, infrastructure). Other transportation is needed during the entire deployment (food, fuel, ammunition, replacements).

Intel Prior to the deployment, but also during the operation, intelligence operations are needed. Examples of prior Intel requirements are: What are the expected enemy forces, what is the available infra structure (communication, resources (water, food etc))? What are safe routes for transportation? What are the local terrain conditions? How is the local climate? What kind of protection is needed for the initial transports? Examples of Intel during the deployment are: What is the enemy behavior? What is the attitude of the local civilians?

C2 & Communication A command and control and a communication infrastructure has to be built-up for the operation: radio, telephone (including GSM), satellite for communication with headquarters and allies including Non Governmental Organisations (NGOs), computer networks for the exchange of information, encryption equipment, Internet for welfare communication.

Procurement Often special equipment and goods have to be procured for the mission. Standard ordering procedures often have to be circumvented because they take too long and ad-hoc procedures have to be used instead.

Protection What kind of weapon and sensor systems have to be used? Are they allowed by the Rules of Engagement (ROE)?

Budget What will be the costs of the deployment? How do we stay within the maximum allowed budget limits?

These processes can be very dynamic, because, for example, Intel information, changing ROE's and political involvement can influence already started tasks. Planning of these complex operations often involves the commitment of large numbers of geographical distributed staff personnel over periods varying from several weeks to several months. Currently, normal communication channels like telephone and e-mail are used for the exchange of information, while in general spreadsheet and database applications are used for maintaining information, mostly on an individual or small departmental basis. This means that other people and departments do not have insight into this information and should make explicit requests (by telephone or e-mail) to obtain it. Moreover, one has to deal with international partners, both military and civil with which plans have to be aligned.

It is clear that dynamic workflow applications can be of great help for planning these operations. We summarize a number of issues that can be supported:

- support of the overall structure of the entire process;

- supplying the right information to the right parties at the right moment. We are dealing with a variable number of dislocated people causing a dynamic topology;
- the automatic checking of deadlines and taking actions in case they are passed;
- interrupting activities already started due to changed circumstances;
- monitoring the status of actions with the possibility to interrupt or reallocate tasks (automatically or by users);
- the ad-hoc creation or adaptation of workflows for subtasks by users;
- automatically monitoring and checking of budget.

8.3.3 Intelligence Operations in Asymmetric Warfare

Intelligence operations are becoming a more and more important part of modern warfare. Especially in counter terrorism the timely gathering of information about plans of adversaries is the most important weapon against them. Many people and systems are involved in this gathering of information. As a result a large amount of information is generated, which easily leads to an information overload and, as a result, important information is often not available at the right moment at the right place. Using a dynamic workflow system like can help to structure the information streams in this information gathering. For example, a local agent may obtain information about a possible adversary. A workflow can now be started and as a first step the user has to enter information in a form. This information can be used by the system to start a workflow that takes care that appropriate actions are taken. Using the automatic monitoring and timeout features of iTask special actions can be taken in case insufficient progress is made.

8.3.4 Crisis-Management and Civil-Military cooperation

In crisis-management operations one is often confronted with situations where people of many different organizations have to cooperate, often in an ad-hoc manner, to tackle the crisis. An important issue here is that these different organization all may have their own command and control procedures and systems (stove pipes). Dynamic workflow applications can be used to integrate information from different systems, and to supply the overall command and control for operations. For these kinds of collaborations it is important that different organization can easily join the system without the need to install special software.

Crisis-Management operations are in general very unpredictable and it is therefore impossible to capture their command and control in a predefined dynamic workflow. But the generic structure of these operations can be captured in a workflow and for many more detailed subtasks dedicated workflows can be defined. Having a tool that allows for using human improvisation to combine a generic structure with detailed workflows for subtasks can be of great help.

8.4 Strengths and Weaknesses

Military operations and civilian, or joint civil-military crisis response operations share many characteristics. Both have to deal with complex resource allocation and complex information management in a potentially hostile environment. Taking into consideration a further convergence of military and joint civil-military operations we view these domains therefore as single broad domain. This raises the bar somewhat in comparison with pure military C2 systems, because existing structures, such as an established chain of command, or a known level of training cannot be assumed to be available.

Because the iTask workflow language is embedded in a general purpose programming language, it can in principle be used to construct any C2 support, crisis response, or other process support system imaginable. However, the required effort that is needed and the quality of the resulting system is determined largely by what is offered out-of-the-box. Therefore, it is important to know whether what is currently offered by the iTask toolkit matches the needs of the domain for which one aims to build systems.

When proposing technological solutions, different authors highlight different requirements as being important (see for example [IEH09] and [SB09]). To determine the status quo of the iTask system's applicability for the joint crisis-management / military domain and to identify areas for future research without bias, we need an independently defined set of requirements. For the crisis-management domain, Jul in [Jul07] provides such a set of five design requirements distilled from an analysis of the domain:

- **Design Requirement #1:** Response technology should seek to support just-in-time learning, first, of the task the tool is intended to support, second, of the needs and goals of the present operation, and, third, of disaster management practices in general.
- **Design Requirement #2:** Response technology, even when focused on agent-driven tasks, should seek to aid response-driven tasks, such as planning, coordination and resource management.
- **Design Requirement #3:** All response technology should actively nurture cooperation, collaboration and partnership formation.
- **Design Requirement #4:** Response technology, while imposing standard structures and procedures, must, insofar as possible, allow flexibility and deviation in their application.
- **Design Requirement #5:** Response technology should aim for *graceful augmentation*, allowing the technology to be integrated in or removed from the user's activities with a minimum of disruption.

In this section we systematically discuss the strengths and weaknesses of the iTask system in view of each of these requirements. The purpose of this discussion is

not to evaluate whether the system, in its current form, is ready and useful for deployment during a crisis response or military operation. Its primary goal is to uncover interesting challenges to focus further research.

8.4.1 Requirement #1: Just-in-time Learning

Because people do not need to know what they will have to do in advance, the step-by-step guidance through standard procedures by a workflow is essentially just-in-time learning of those procedures. The workflow specification guides people through procedures they might have never done before. Once users learn how to use the interface to find out what tasks they have to do, and how they can select tasks to work on, they can rely on the system to tell them what needs to be done. To ease the initial learning curve, the iTask user interface has been designed to resemble an e-mail client as much as possible. Users can simply think of the system as a special e-mail system where all messages in their inbox happen to be requests to do something.

A weakness of the iTask system is that the goals and instructions of tasks are communicated primarily through text as defined in the workflow models. When a user is presented with a task having instructions he or she cannot understand, or even worse, can misunderstand, there are no built-in ways to easily resolve that knowledge gap. The learnability of the tasks is therefore almost completely determined by the degree to which the workflow models supply enough information. Of course, this problem also exists for paper handbooks and contingency plans. Interactive workflow systems have an opportunity to do more, e.g. to provide access to information sources, or to provide easy communication to ask peers help. Currently, the iTask system does not yet offer any support for learning at the task level.

8.4.2 Requirement #2: Response Driven Tasks

A workflow system, by definition, supports response driven tasks, since its sole purpose is to automate the coordination and execution of standard procedures. It has the additional advantage over hard-coded support systems of having inspectable models at run-time that can be queried to get information about what is going on. The dynamic data-driven workflow models that are used by the iTask system have the additional potential of enabling flexible resource allocation and planning. Data that becomes available as a result of performed tasks can be used for the (re)distribution of resources or for planning/scheduling of other tasks. However, currently available resource allocation combinators in the iTask system's standard library are purely algorithmic. It is possible to integrate stochastic or other predictive models to distribute tasks and resources, or to support decision making at crucial points in a workflow. Having such tasks available in a library of the workflow language could further improve the support of response driven tasks.

8.4.3 Requirement #3: Cooperation and Collaboration

Cooperation and collaboration are supported in iTask workflow models by (re) assigning tasks to users and routing the task results from one user to another. When tasks are delegated to others, the user who delegated them can track them. It is also possible to define workflows that add new users to the system, who then immediately can get tasks assigned to them.

The multi-user features of the iTask system make it possible to define workflow models that stimulate the involvement of multiple users. However, to assume that therefore it *"nurtures cooperation, collaboration and partnership formation"* would be too shortsighted. There are still many things that should be facilitated to promote cooperation, regardless of the concrete tasks at hand, such as for example, integrated communication capabilities (chat, voice, video) to enable users to discuss the tasks they are working on, or formation of ad-hoc teams of users.

A more fundamental property of the iTask system that influences the possibility of defining "cooperation friendly" workflow specification is the focus on users as individuals. Tasks are always assigned to, and managed by, a single individual. Social relations, both formal and informal, between users are not modeled in the iTask system. In daily life, however, it is not uncommon to work together on a task without exactly dividing it into discrete subtasks, or to have shared responsibility for a task.

A final issue is the ability to cooperatively define and plan tasks. By default, iTask workflow models are controlling tasks in a top-down manner, assigning tasks to users as planned in a workflow specification. However, because tasks can be results of other tasks and tasks can receive tasks as their input, it is possible to define meta-workflows that let users agree upon a set of tasks and their order to define a new workflow.

8.4.4 Requirement #4: Flexibility

Flexibility is a feature of the iTask system that pointed us to the potential usefulness of dynamic workflows for crisis management in the first place. Because iTask workflow specifications support the modeling of dynamic processes at three different levels, as explained earlier, it is potentially capable of complete compliance with this fourth requirement.

However, we should not claim victory too soon. Although it is technically possible to define very flexible workflow models, we must acknowledge that the usefulness of this expressive power is constrained by the interface through which it is exposed to end-users. Additional research is needed to develop generically applicable problemsolving patterns that can be applied when normal procedures do not apply. More research is also needed on what information is required by users to become aware that there is a need for deviation from standard procedures, and what is required to decide what course of action is to be selected to resolve the issue.

8.4.5 Requirement #5: Graceful Augmentation

Meeting this final requirement completely is near impossible for any workflow system because removal of a workflow system during the execution of an operation guided by it will cause disruption. It is possible to meet this requirement as closely as possible by reducing the amount of disruption if (a part of) the system is temporarily removed. The current iTask system does not specifically address this issue, because network infrastructure has been assumed to be available. However, it has been shown that it is possible to run parts of workflows offline (see [PJKA08] or chapter 6), by transferring part of the workflow computation to the client system. It is, of course, possible to use the system in a controlled environment such as a command post while communicating tasks through other channels, where it could still have advantages over written handbooks, because iTask workflow specifications are dynamic.

8.4.6 Additional opportunities

Because the requirements suggested by Jul cover crisis response technology in a very broad sense, there are properties of the use of dynamic workflow models to support operations that cannot be linked directly to one of the requirements, but are potentially valuable. Examples include:

- **Verification through formalization and prototyping:** In written plans and procedures, anything can be described, even when logically contradictory, ambiguous, or otherwise incorrect. By formalizing workflows in a modeling formalism, one is forced to write down precisely what the steps in the process are. But even then, workflows can be defined that are logically sound, but nonetheless make no sense at all. Because the iTask system can generate executable systems instantly from models, it is possible to rapid prototype workflow models and to verify them through simulation and testing during the design phase.

- **Data to create situation awareness:** When processes and actions are coordinated and communicated through a workflow support system, there is an abundance of data available during operations about what tasks people are working on and what processes are currently running. This data, when presented in the right way to the right people, could be valuable for assessing the situation and for planning further action. Although this data is unavoidably incomplete and it is not immediately clear how to extract useful information from it, it offers interesting opportunities. Further research is needed.

- **Data for evaluation and learning:** The availability of data about tasks and processes is not only useful during operations, but may also be utilized afterwards to evaluate an operation and learn from the mistakes that were made.

8.5 Future Challenges

From the discussion of the iTask system in the previous section we can conclude that, although there is substantial potential that certainly justifies further research, it is not the perfect programming toolkit for building C2 or crisis-management systems yet. There are still challenges that have to be tackled.

8.5.1 Collaboration

One area where the iTask system could gain greatly is in the facilitation of collaborative work. The current focus on individual users, without the concept of (informal) organizations, limits collaboration or partnership formation. The communication through formal task assignment only also limits its potential.

Quick wins can be achieved by integrating easy-to-build communication features such as chat sessions linked to tasks or enabling users to let others view the tasks they are working on. Although this would make it easier to get help with, or give feedback on tasks, a much bigger challenge lies in the integration of social constructs like organizations, (temporary) teams, partnerships or friends. This would decouple the direct relation between a task and an individual person and raises questions about dealing with concurrency, shared responsibility, shared decision making and individuals performing tasks on behalf of organizations.

Another way of facilitating the creation of cooperation friendly workflow models could be the development of out-of-the box meta-workflows for collaboratively defining and assigning tasks.

8.5.2 Effective Flexibility

Another challenge for iTask's design would be to apply the power of adapting running processes to resolve unexpected problems that arise during operations. Although it is technically possible to adapt workflows that are already running, two important questions that would have to be answered are: First, how will users know that the workflow they are executing is not going to fulfil its goal? And second, how should they instruct the system to change the workflow to resolve the problem?

To answer the first question, more research is required into what information about a workflow instance is needed by users to be able to detect that there is a problem. A related issue is whether it is possible to monitor progress automatically and to warn users of an unexpected lack of progress.

The second question is possibly even more challenging. An easy way out would be to let end-users solve the problem by providing some (visual) programming interface to specify alternative workflows. However, this assumes that all users can, and want to use this when facing an immediate problem. We believe the bigger challenge is the design of an interface to the underlying workflow model that helps stranded users in either resolving their immediate problems and continue with minimum disruption, or let them gracefully abort.

8.5.3 Domain Specific Frameworks

The design of workflow specifications is likely to be influenced by which basic tasks, combinators and generic sub-processes are readily available. For example, if there are meta-workflows supporting collaborative task assignment available in a library, it is more likely that collaborative steps will be incorporated in a workflow than when the collaboration process itself also has to be specified. It is therefore necessary to have available a collection of tasks, data types and generic workflows that are common in the domain. A major challenge will be the design of a domain-specific framework that supplements the generic iTask system to create a platform for building workflow support systems to aid crisis-management operations.

8.6 Conclusions

In this paper we presented dynamic workflow programming, as implemented by the iTask system, as a candidate platform for developing applications to support command and control of military and crisis-management operations. Because of its unique features like: data driven, parametrizable workflows and extensive support of dynamic behavior we view it as a potentially valuable tool for construction of C2 systems for this domain. We have explained the basics of programming such systems using the iTask toolkit, and sketched our vision of possible applications in the military and crisis- management domains, which we consider a single domain in this context. Most notably, we have compared the current iTask system to independently defined requirements for technology during crises defined by Jul in [Jul07]. We have discussed the strengths and weaknesses of the iTask system in light of these requirements to identify future research challenges. Based on this comparison, we are confirmed in view that dynamic workflow programming is indeed potentially valuable, but research challenges are: facilitation of collaboration on tasks, interaction with the workflow model during execution, and the need for domain specific frameworks. By focusing research effort on these issues, we hope to develop the system further, into a valuable C2 construction toolkit for building systems that flexibly support people under demanding circumstances.

Chapter 9

Conclusions and Discussion

The main goals of research for this thesis were: the extension of the iTask system with client-side processing under the condition that the declarative nature of the iTask system should be maintained; the realization of a client-side execution platform for this using a dedicated interpreter; an investigation of the possibilities to use the iTask system for applications in the area of military and crisis-management operations.

We showed that the first two goals could be realized by developing the Sapl interpreter for use at the client side, extending the Clean back-end with a Sapl generator and extending Clean Dynamics to Clean-Sapl Dynamics. Clean-Sapl Dynamics adds the possibility to serialize functions at the server side to move them to the client and to use them there. In this way the technique that was used for local task-tree rewriting could directly be generalized for client-side task-tree rewriting. Moving a task from server to client only required the addition of the `OnClient` annotation to the task. For the client-side Clean platform we developed an elementary intermediate functional language using a minimum of concepts and implemented an efficient interpreter for this language.

In the next sections we reflect on our work, look at related work, discuss a number of issues that came up after the papers presented in the previous chapters were finished and sketch future research for (applications of) the iTask system. For this we use the same division into three parts as we did in the introduction.

9.1 Part 1: Formalism and Implementation of Functional Languages

The first part of this thesis dealt with a functional programming formalism and an implementation for this formalism.

Chapter 2: The Importance of a Compact Programming Formalism

In this chapter we described how the Scott encoding used for representing ADT's can also be used for describing algorithms in the λ-calculus in a comprehensive

way. Of course, this work is not on the main track of this thesis, and more or less inspired by the authors obsession for finding a minimal programming formalism. Finding such a formalism is of both practical and philosophical importance. A compact and comprehensive formalism is an ideal subject for study, education and theoretical considerations. From a philosophical point of view it is important to have a formalism that is both comprehensive and based on as few concepts as possible. The λ-calculus is such a formalism. Using the Scott encoding for the representation of algebraic data types, turns the λ-calculus into a better comprehensible and more efficient formalism for the expression of algorithms.

Chapter 3 and 4: Interpreters and Compilers for Functional Programming Languages

The basic SapI programming language is based on the operation of function application only and can therefore be implemented using a pure graph-reduction interpreter. This interpreter can therefore be considered as a model implementation of functional programming languages using graph reduction. It is therefore very suitable for text books and introductory courses on the implementation of functional programming languages.

Evaluation of Efficiency of the SapI Interpreter

The SapI interpreter is faster than a number of other interpreters for lazy functional programming languages. In chapter 3 we discussed a number of reasons for this better performance. Recently, we did some more experiments with the interpreter and gained a better understanding of the reasons for its efficiency. The implementation of SapI is based on simple graph-reduction techniques. Basically, graph reduction consists of two steps: instantiation and reduction of graphs. During instantiation the graph representing the body of a function is (partially) copied and variables in the body are replaced by arguments on the stack. During reduction the resulting graph is destructed (reduced) and arguments are pushed on the stack to be used by a new instantiation. Graph instantiation is the most expensive operation in the SapI interpreter. Therefore, making this operation as efficient as possible is important. The selective instantiation optimization used in SapI results in smaller graphs to be instantiated. SapI also uses a more compact representation of function applications with 1 or 2 arguments. Another important issue is making (de)allocation of graph nodes as cheap as possible. For this the SapI interpreter uses a strongly simplified memory management method. After starting the interpreter SapI allocates a fixed (user defined) amount of memory and only returns this memory to the operating system after closing the interpreter. Most compiled applications use memory in an incremental way, starting with a relatively small amount of allocated memory and increasing this amount when necessary. The fixed amount of allocated memory for SapI is less a problem because this amount is mostly small (10-50 Mb for typical SapI applications) in comparison with the total amount of memory available. Memory consists of an array of cells (graph nodes). Cells are allocated one-by-one during

graph instantiation and block instantiation is not supported. Therefore, fragmentation of memory is not an issue. SapI uses a straightforward mark-and-sweep garbage collector. Each cell has a gc bit, which is set on allocation and inverted during marking. There is no explicit reclaim of cells. During allocation the interpreter just searches for the next free cell in memory. As a result memory management is simple and efficient, with very little overhead, which reduces the price of the graph reduction implementation considerably.

Improving the Performance of the SapI Interpreter

We also did some experiments trying to further improve the performance of the SapI interpreter. For compilers for lazy functional programming languages the standard way to optimize is to avoid the use of graph reduction as much as possible. Strictness analysis is the most frequently used technique for finding out whether graph reduction can be omitted (see [PvE93] and [PJ87]). Whenever possible, graph-reduction code is replaced by code comparable to code generated by compilers for imperative programming languages. In this way significant speed-ups (up to a factor of 40) can be obtained (see also chapter 4).

What would be the gain if similar techniques are applied to the SapI interpreter? We did some experiments and implemented a small interpreter for pure numeric functions and expressions. For such an interpreter we basically have two possibilities. Either we use an interpreted virtual machine with an own instruction set and generate code for this machine, or we make an interpreter by transferring a tree representation of the program using a stack like structure for doing calculations, storing intermediate results and function arguments. In chapter 3 we argued that a virtual machine approach with low level byte instructions is not likely to be efficient. Therefore, we implemented the tree transfer approach using the same tree representation as was used in the graph-reduction implementation. For this implementation only minor speed-ups in comparison with the graph-reduction approach could be obtained (at most a factor of 2 in very special cases, and in general less than 10-20%). This may seem very surprising in comparison with the speed-ups that can be obtained for compilers. But these speed-ups can only be obtained because optimal use can be made of the architecture of the processor using registers and stack to do all computations. For an interpreter the interpretive overhead now becomes a dominant factor.

We also made use of tail recursion optimization in the SapI compiler (see chapter 4). This resulted in speed-ups up to a factor of 7 with respect to the basic SapI compiler. For an interpreter tail recursion can be optimised too. But again, the obtained speed-up is marginal, at most 20 to 30% in the most optimal cases. The greatest gain of this optimisation is a reduction in the use of stack space and the possible prevention of stack overflows.

A better way to obtain significant speed-ups for an interpreter is to take an opportunistic approach. We can use the SapI compiler to compile all standard libraries (especially those dealing with lists). The generated code can be made part of the SapI interpreter. It is even possible to hand optimize the generated C++

functions before adding them. SapI programs making use of these functions will now run more efficiently. The only price to pay is a larger executable for the SapI interpreter. This approach is not uncommon for interpreters. Amanda [Brub] also uses this optimization extensively.

The use of SapI's Scott encoding by others

The encoding of ADT's used in SapI and the efficency of the SapI interpreter inspired Matthew Naylor and Colin Runciman in [NR08] to make an FPGA (Field Programmable Gate Array) implementation for the Yhc Core intermediate language. They also use the Scott encoding of ADT's and therefore only need to implement graph reduction and primitive operations on basic types. In this way a very simple implementation that is suitable for realisation on FPGA's can be made. A performance bottleneck in the operation of graph instantiation on custom hardware is the sequential copying of a graph (what they call the von Neumann bottleneck). On an FPGA this instantiation can be done in parallel for greater chunks. In this way a significant speed-up in comparison with sequential instantiation can be obtained.

9.2 Part 2: The iTask system and Client-side Processing

In this part we described the iTask system and the realisation of client-side processing for this system.

Chapter 5: iTask as an Embedded Language

The iTask system is an example of a Domain Specific Language (DSL) embedded in another language. As argued in chapter 5, a functional language like Clean is very well suited for the embedding of such a DSL. The combination of combinators, generic type driven functions, Dynamics and higher order functions make it possible to realize this. The resulting formalism is a real extension of Clean, offering functionality that is hard to program directly in Clean (without the use of such a combinator library), or any other programming formalism. The development of a system like iTask from scratch would be a tremendous effort. Arthur Baars in his PhD thesis [Baa09] also advocates the use of functional programming languages for the embedding of DSLs.

This all shows that modern lazy functional programming languages like Haskell and Clean are more than just programming languages. They are actually tool building languages. With them one can realize new programming platforms in only a fraction of the time that would be needed to implement these platforms from scratch. But Haskell and Clean were originally not designed for this purpose. This is reflected in a number of things. Type error message are given at the level of the host language and are often incomprehensible for the user of the DSL. The syntax of the DSL has to follow the syntax of the host language. For the DSL user this

sometimes results in a rather awkward syntax. Some Haskell pre-processors offer syntax macro's to relieve this problem a bit, but this also reinforces the problem of incomprehensible type error messages. An interesting object for research is the design and development of a dedicated language extension of Clean or Haskell especially intended for embedding domain specific languages in. This language should allow for a dedicated syntax for the DSL and dedicated type error messages.

Chapter 6 and 7: Client-Side Processing

Clean-Sapl Dynamics (an extension of Clean Dynamics) is used for moving execution from server to client. Clean-Sapl Dynamics is also used for the addition of dedicated call back functions to plug-ins in the iTask system that could be executed on either server or client. Clean Dynamics also plays an important role in the implementation of the iTask system itself, including the recently added *Change* [PAK+10] extension. Therefore, we may conclude that the concept of Dynamics in Clean plays a decisive role in the implementation of the iTask system. Dynamics are a unique feature of Clean that cannot be found in other functional languages.

We used a Java version of the Sapl interpreter for client-side processing. This version is slightly (between 10-50%) slower than the C++ version. The main reason for this is the use of extra indirections using arrays to mimic pointer manipulations. It is, of course, possible to make a browser plug-in directly in C++. But this requires different plug-ins for different browsers. Using the Java applet plug-in, which is available for all popular browsers, only one Java implementation is needed. A small disadvantage of using a plug-in is that the plug-in must be loaded into the web-browser. This usually takes a few seconds for Java applets.

An interesting alternative for the use of plug-ins is the use of JavaScript as a platform for client-side execution. A JavaScript interpreter is standard included in all popular browsers. Hence it is not required to load plug-ins, which saves time and does not require the availability of a plug-in for each execution platform. Nowadays JavaScript is used extensively for the implementation of client-side processing. Therefore, the efficiency of the JavaScript interpreters has increased significantly during the last years. Just-In-Time (JIT) compilation techniques are used to achieve this. This makes JavaScript worthwhile to consider as a platform for client-side execution. We have two alternatives for using JavaScript at the client side. First, we can use JavaScript to implement a Sapl like interpreter. Second, we can generate JavaScript using an dedicated version of the Sapl compiler. Both approaches will be explored in the near future. Curry [Han07] and HOP [SGL06] are examples of approaches that use JavaScript generation from functional languages. But these are strict languages, making it relatively easy to generate efficient JavaScript code, something which is much more difficult for a lazy language like Clean. JavaScript generation will certainly be a useful candidate for generating simple functions, for example for checking form input. These functions are often strict and can easily be translated to efficient JavaScript. For non-strict functions we expect the efficiency for generated JavaScript to be lower than for their Java interpreted counterparts, but more research is needed to give a definite answer.

New developments for the iTask System

The iTask system is still developing. The system has been restructured and is now implemented as a server application offering a set of web services (see [LP09b]). Also a new web client for iTask is implemented, making use of the ExtJS framework [Ext]. All form generation is now done at the client side and all data exchange between server and client is based on exchanging data generated by generic (type driven) functions.

Another important extension is the *Change* concept [PAK+10]. Using *Change* workflows can be adapted on-the-fly in a number of ways. First, tasks can be replaced by other tasks that are (if necessary) loaded into the application using Clean Dynamics. Second, users can construct new tasks such as simple forms and make simple compositions using existing or just created tasks. The concept of *Change* is absolutely necessary to build useful systems for domains like military and crisis-management operations. Although, much of the underlying technical machinery is already available, further research is needed for offering the end-user the right user interface providing the information needed for changing and constructing tasks. This will be a main line of research for the near future.

In [PAK+10] also an executable operational semantics of a core version of the iTask system is given (this is an extension and improvement of the earlier versions [KPA09] and [AEMP08]). The operational semantics models iTask as a term rewriting system that reduces terms according to input events produced by workers. In [PAK+10] this operational semantics is used to give a meaningful implementation and semantics of the *change* operation in iTask. We have planned to extend the semantics in such a way that it can also be used for a more formal definition of local and client-side task-tree rewriting.

The iTask system is more than just a library for building dynamic workflow applications. The basic tasks in the system could be any process that edits (changes) or yields data. In fact the iEditors from chapter 7 are a first example of such a dedicated task. In [LP09b] other examples of such tasks are mentioned, e.g. a Python client that monitors a file system for new documents and automatically starts a new workflow with the document as attachment, or the integration of GoogleMaps in tasks. Also the restriction to Internet browsers for the execution of tasks is artificial. Tasks can be executed on any computer connected in a network. Therefore, iTask can also be considered as a multi-user and multi-process co-ordination or application structuring tool.

9.3 Part 3: Applications of the iTask System

In the last part of this thesis we discussed possible application areas for the iTask system.

Chapter 8: Applications for Military and Crisis-Management Operations

We have chosen military and crisis-management operations as the main application areas for iTask. These operations are characterized by their unpredictability and need for dynamic on-the-fly adaptability. The dynamic and flexible aspects of iTask make it a logical candidate to be used in these areas. As mentioned above, a good implementation of the *Change* concept will be essential for successful use in these areas. Applications in the military and crisis-management domains and further research into iTask itself will be the subject of a number of follow-up projects.

Alternative Applications for iTask in the military domain

iTask also has potential for other applications in the military domain. One of them is Command and Control of military operations that also involves complex coordination of actions and the integration of sensor and weapon systems. It is clear that also these operations are candidate for support by flexible workflows systems like iTask. But in these cases there is an extra challenge because integration with the already very complex and real-time command and control systems is necessary.

Another area is maintenance of military equipment. Current maintenance procedures are supported by standard Enterprise Resource Planning (ERP) applications. Maintenance is performed on a regular basis (every x months or every y kilo-meters) without taking into account the actual use of the systems. There is a tendency to make maintenance more depending on the actual use of systems and to make maintenance procedures less disruptive. This requires flexible collection of information about use of systems and flexible execution of maintenance procedures. A dynamic workflow system like iTask seems to be a good candidate to support this. Discussions with experts on these subjects already started and will be continued in the near future.

9.4 Final Conclusions

This thesis covers a wide range of topics within the field of functional programming, varying from (minimal) programming formalisms, their implementation and applications of functional programming languages in the Internet domain. The research resulted in the realization of a programming formalism and an efficient execution platform for this formalism to be used for client-side processing in the Internet domain. It shows how advanced concepts like Dynamics in Clean are extended and used for smooth, seamless, moving of execution from server to client.

The iTask dynamic workflow system constituted the most important application area for these results. The iTask toolkit offers a new way for structuring web applications and modeling of complex business processes in various domains. The concept task in the iTask system offers a powerful modeling concept as well as a natural starting point for the implementation of client-side processing, the inclusion of plug-ins, etc. The current version of the iTask system is powerful enough to

construct usable prototype workflow systems. These systems endorse the power of the data centered approach and provides important information for future research. Challenges to be tackled are the collaboration on tasks and dynamic changes of the workflow. The further development of this system and its applications will remain an important research area for the near future.

This thesis confirmed that modern functional programming languages are a perfect tool for the construction of embedded domain specific languages. With limited effort one can make advanced domain specific languages. The availability of dynamics and generics proved to be powerful ingredients. The full range of possibilities of these languages is still not fully explored!

Bibliography

[ABE05] Birger Andersson, Maria Bergholtz, and Ananda Edirisuriya. A Declarative Foundation of Process Models. In Oscar Pastor and João Falcão e Cunha, editors, *Proceedings 17 Int'l Conference on Advanced Information Systems Engineering, CAiSE 2005*, volume 3520 of *LNCS*, pages 233–247. Springer-Verlag, 2005.

[Ach07] Peter Achten. Clean for Haskell98 programmers - A quick reference guide. `http://www.st.cs.ru.nl/papers/2007/achp2007--CleanHaskellQuickGuide.pdf`, July 2007.

[AEMP08] Peter Achten, Marko Eekelen, Maarten de Mol, and Rinus Plasmeijer. A common Arrow based semantics for GEC and iData applications. Technical Report ICIS–R08023, Institute for Computing and Information Sciences, Radboud University Nijmegen, The Netherlands, December 2008.

[AGS99] D.S. Alberts, J.J. Garstka, and F.P. Stein. *Network centric warfare: Developing and leveraging information superiority.* C4ISR Cooperative Research Program Publications Series, Department of Defense, 1999.

[AHKB02] Wil van der Aalst, Arthur ter Hofstede, Bartek Kiepuszewski, and Ana Barros. Workflow patterns. QUT technical report, FIT-TR-2002-02, Queensland University of Technology, Brisbane, Australia, 2002.

[AP02] Artem Alimarine and Rinus Plasmeijer. A generic programming extension for Clean. In Thomas Arts and Markus Mohnen, editors, *Selected Papers of the 13th International Symposium on the Implementation of Functional Languages, IFL '01*, volume 2312 of *Lecture Notes in Computer Science*, pages 168–186, Stockholm, Sweden, September 2002. Springer-Verlag.

[Aug85] L. Augustsson. Compiling pattern matching. In Jouannaud, editor, *Conference on Functional Programming Languages and Computer Architectures, Nancy*, volume 201 of *Lecture Notes in Computer Science*, pages 368–381. Springer Verlag, 1985.

[Baa09] Arthur Baars. *Embedded Compilers.* PhD thesis, University of Utrecht, 2009. ISBN 978-90-8891-131-6.

[Bar84] Henk Barendregt. *The lambda calculus, its syntax and semantics (revised edition)*, volume 103 of *Studies in Logic*. North-Holland, 1984.

[Bar97] H.P. Barendregt. The impact of the lambda calculus in logic and computer science. *The Bulletin of Symbolic Logic*, 3(2):181–215, 1997.

[BB85] C. Bohm and A. Berarducci. Automatic synthesis of typed λ–programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[BB93] A. Berarducci and C. Bohm. *A self-interpreter of lambda calculus having a normal form*, volume 702 of *Lecture Notes in Computer Science*, pages 85–99. Springer Verlag, 1993.

[BCC07] Marco Brambilla, Jordi Cabot, and Sara Cornai. Automatic Generation of Workflow-Extended Domain Models. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Proceedings Model Driven Engineering Languages and Systems, 10th Int'l Symposium, MoDELS 2007*, volume 4735 of *LNCS*, pages 375–389. Springer-Verlag, 2007.

[BMS80] R. M. Burstall, D. B. MacQueen, and D. T. Sannella. Hope: An experimental applicative language, 1980.

[BPSM98] Tim Bray, Jean Paoli, and C.M. Sperberg-Macqueen. Extensible Markup Language (XML) 1.0 (w3c recommendation), 1998. Technical Report, `http://www.w3.org/TR/1998/REC-xml-19980210`.

[Brua] D. Bruin. Personal communication. Noordelijke Hogeschool Leeuwarden, the Netherlands.

[Brub] D. Bruin. The amanda interpreter. `http://home.kpn.nl/janmartinjansen/amanda204.zip`.

[CHS72] H. Curry, J. Hindley, and J. Seldin. *Combinatory Logic*, volume 2. North-Holland Publishing Company, 1972.

[CLWY06] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: web programming without tiers. In *Proceedings of the 5th International Symposium on Formal Methods for Components and Objects, FMCO '06*, volume 4709, CWI, Amsterdam, The Netherlands, 7-10, November 2006. Springer-Verlag.

[CLWY07] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. An idiom's guide to formlets. Technical report, The University of Edinburgh, UK, 2007. `http://groups.inf.ed.ac.uk/links/papers-/formlets-draft2007.pdf`.

[EBJ06] M.R. Endsley, B. Bolte, and D.G. Jones. *Desining for sitation awareness: An approach to user centered design*. Taylor & Francis, London, 2006.

[ER97] E Evans and D Rogers. Using Java applets and CORBA for multi-user distributed applications. *Internet Computing, IEEE*, 1:43–55, 1997.

[Ext] ExtJS framework. `http://www.extjs.com`.

[Fuj] Fujitsu. iFlow. `http://www.fujitsu.com/interstage`.

[FW08] Dirk Fahland and Heiko Woith. Towards process models for disaster response. In M. de Leoni, S. Dustdar, and A.t. Hofstede, editors, *Proceedings of the First International Workshop on Process Management for Highly Dynamic and Pervasive Scenarios (PM4HDPS), co-located with 6th International Conference on Business Process Management (BPM'08)*, September 2008.

[Gar05] J.J. Garrett. Ajax: A new approach to web applications, 2005. `http://www.adaptivepath.com/ideas/essays/archives/000385.php`, visited March 2009.

[GHC] *The GHC g*. `http://www.Haskell.org/ghc`.

[GJS96] J Gosling, B Joy, and G Steele. *The Java Language Specification*. Sun Microsystems, 1996.

[Han07] Michael Hanus. Putting declarative programming into the web: translating Curry to JavaScript. In *Proceedings of the 9th International Conference on Principles and Practice of Declarative Programming, PPDP '07*, pages 155–166. ACM Press, 2007.

[Hin00] Ralf Hinze. A new approach to generic functional programming. In *Proceedings of the 27th International Symposium on Principles of Programming Languages, POPL '00*, pages 119–132. Boston, Massachusetts, USA, January 2000.

[Hin05] Ralf Hinze. Theoretical pearl Church numerals, twice! *Journal of Functional Programming*, 15(1):1–13, 2005.

[Hug] *Hugs Online*. `http://www.Haskell.org/hugs`.

[HVV08] Zef Hemel, Ruben Verhaaf, and Eelco Visser. WebWorkFlow: an object-oriented workflow modeling language for web applications. In K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter, editors, *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems, MoDELS '08*, volume 5301 of *LNCS*, pages 113–127. Springer-Verlag, 2008.

[IBM] IBM. IBM WebSphere MQ. `http://www.ibm.com/software/integration/wmq`.

[IEH09] Magnus Ingmarsson, Henrik Eriksson, and Niklas Hallberg. Exploring development of service-oriented c2 systems for emergency response. In J. Landgren and S. Jul, editors, *Proceedings of the 6th International ISCRAM Conference - Gothenburg, Sweden*, May 2009.

[JKP06] Jan Martin Jansen, Pieter Koopman, and Rinus Plasmeijer. Efficient interpretation by transforming data types and patterns to functions. In Henrik Nilsson, editor, *Revised Selected Papers of the 7th Symposium on Trends in Functional Programming, TFP '06*, volume 7, pages 73–90, Nottingham, UK, 2006. Intellect Books.

[JKP08a] Jan Martin Jansen, Pieter Koopman, and Rinus Plasmeijer. From interpretation to compilation. In Zoltán Horváth, editor, *Proceedings of the 2nd Central European Functional Programming School, CEFP '07*, volume 5161 of *Lecture Notes in Computer Science*, pages 286–301, Cluj Napoca, Romania, 23-30, June 2008. Springer-Verlag.

[JKP08b] Jan Martin Jansen, Pieter Koopman, and Rinus Plasmeijer. Web based dynamic workflow systems and applications in the military domain. In Theo Hupkens and Herman Monsuur, editors, *Netherlands Annual Review of Military Studies - Sensors, Weapons, C4I and Operations Research*, pages 43–59, 2008.

[JLP10] Jan Martin Jansen, Bas Lijnse, and Rinus Plasmeijer. Towards dynamic workflows for crisis management. In Mark Haselkorn and Simon French, editors, *Proceedings of the 7th International Conference on Information Systems for Crisis Response and Management, ISCRAM '10*, Seattle, WA, USA, May 2010. To appear.

[JLPG10] Jan Martin Jansen, Bas Lijnse, Rinus Plasmeijer, and Tim Grant. Web based dynamic workflow systems for C2 of military operations. In *Revised Selected Papers of the 15th International Command and Control Research and Technology Symposium, ICCRTS '10*, Santa Monica, CA, USA, 2010. To appear.

[JPK09] Jan Martin Jansen, Rinus Plasmeijer, and Pieter Koopman. iEditors: extending iTask with interactive plug-ins. In Sven-Bodo Scholz, editor, *Revised Selected Papers of the 20th International Symposium on the Implementation and Application of Functional Languages, IFL '08*, 2009. To appear in Springer LNCS 5836.

[JPK10] Jan Martin Jansen, Rinus Plasmeijer, and Pieter Koopman. Functional pearl: Comprehensive encoding of data types and algorithms in the λ-calculus. *Journal of Functional Programming*, Submitted for publication, 2010.

[JPKA10] Jan Martin Jansen, Rinus Plasmeijer, Pieter Koopman, and Peter Achten. Embedding a web-based workflow management system in a

functional language. In Claus Brabrand and Pierre-Etienne Moreau, editors, *Proceedings 10th Workshop on Language Descriptions, Tools and Applications, LDTA '10*, pages 79–93, Paphos, Cyprus, March 27-28 2010.

[JSO] *Introducing JSON.* `http://www.json.org`, visited March 2009.

[Jul07] Susanne Jul. Who is Really on First? A Domain-Level User, Task and Context Analysis for Response Technology. In (B. Van de Walle, P. Burghardt, and C. Nieuwenhuis, editors, *Proceedings of the 5th International ISCRAM Conference - Delft, the Netherlands*, May 2007.

[Klu04] W. Kluge. *Abstract Computing Machines.* Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[Kna03] Frederico Caldeira Knabben. FCK editor, 2003. `http://www.fckeditor.net`, visited March 2009.

[KPA09] Pieter Koopman, Rinus Plasmeijer, and Peter Achten. An executable and testable semantics for iTasks. In Sven-Bodo Scholz, editor, *Revised Selected Papers of the 20th International Symposium on the Implementation and Application of Functional Languages, IFL '08*, volume 5836 of *Lecture Notes in Computer Science*, Hertfordshire, UK, 2009. Springer-Verlag. To appear.

[Kri07] Shriram Krishnamurthi. The Flapjax site, 2007. `http://www.flapjax-lang.org`, visited March 2009.

[Lan66] P. J. Landin. The next 700 programming languages. *Commun. ACM*, 9(3):157–166, 1966.

[Lei03] D. Leijen. *The λ Abroad – A Functional Approach to Software Components.* PhD thesis, Department of Computer Science, Universiteit Utrecht, The Netherlands, 2003.

[Lia99] S. Liang. *Java Native Interface: Programmer's Guide and Reference.* Addison-Wesley Longman Publishing Company, 1999.

[LP09a] Bas Lijnse and Rinus Plasmeijer. Between types and tables - Using generic programming for automated mapping between data types and relational databases. In Sven-Bodo Scholz, editor, *Revised Selected Papers of the 20th International Symposium on the Implementation and Application of Functional Languages, IFL '08*, 2009. To appear in Springer LNCS 5836.

[LP09b] Bas Lijnse and Rinus Plasmeijer. iTasks 2: iTasks for End-users. In Marco T. Morazán, editor, *Proceedings 21st International Symposium on the Implementation and Application of Functional Languages, IFL '09*, September 23-25 2009. To appear in Springer LNCS.

[LS07] Florian Loitsch and Manuel Serrano. Hop client-side compilation. In *Proceedings of the 7th Symposium on Trends in Functional Programming, TFP '07*, pages 141–158, New York, NY, USA, 2-4, April 2007. Interact.

[McC98] Glen McCluskey. Using Java reflection, 1998. `http://java.sun.com/developer/technicalArticles/ALT/Reflection`, visited March 2009.

[MF00] E. Meijer and S. Finne. Lambada: Haskell as a better Java. In *Proceedings of the 2000 Haskell Workshop, Montreal, Canada*, 2000.

[MLH99] Erik Meijer, Daan Leijen, and James Hook. Client-Side Web Scripting with HaskellScript. In *Practical Aspects of Declarative Languages, First International Workshop, PADL '99, San Antonio, Texas, USA, January 18-19, 1999, Proceedings*, Lecture Notes in Computer Science, pages 196–210. Springer, 1999.

[Mog94] Torben Ae. Mogensen. Efficient Self-Interpretation in Lambda Calculus. *Journal of Functional Programming*, 2:345–364, 1994.

[NR08] Matthew Naylor and Colin Runciman. The Reduceron: Widening the von Neumann bottleneck for Graph Reduction using an FPGA. In *Implementation and Application of Functional Languages: 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers*, pages 129–146, Berlin, Heidelberg, 2008. LNCS no. 5083, Springer-Verlag.

[OMG96] OMG: Object Management Group. *The Common Object Request Broker: Architecture and SpecificationRevision 2.0*, 1996. `http://www.omg.org/corba-e`, visited March 2009.

[PA] Pallas-Athena. FLOWer. `http://www.pallas-athena.com`.

[PA06a] Maja Pesic and Wil van der Aalst. A declarative approach for flexible business processes management. In Johann Eder and Schahram Dustdar, editors, *Proceedings of the 1st Business Process Management Workshop on Dynamic Process Management, DPM '06*, volume 4103 of *LNCS*, pages 169–180. Springer-Verlag, 2006.

[PA06b] Rinus Plasmeijer and Peter Achten. The implementation of iData - A case study in generic programming. In Andrew Butterfield, Clemens Grelck, and Frank Huch, editors, *Revised Selected Papers of the 17th International Symposium on the Implementation and Application of Functional Languages, IFL '05*, number 4015 in Lecture Notes in Computer Science, pages 106–123, Dublin, Ireland, 19-21, September 2006.

[PAK07] Rinus Plasmeijer, Peter Achten, and Pieter Koopman. iTasks: executable specifications of interactive work flow systems for the web. In *Proceedings of the 12th International Conference on Functional Programming, ICFP '07*, pages 141–152, Freiburg, Germany, 1-3, October 2007. ACM Press.

[PAK08a] Rinus Plasmeijer, Peter Achten, and Pieter Koopman. An introduction to iTasks: defining interactive work flows for the web. In *Revised Selected Lectures of the 2nd Central European Functional Programming School, CEFP '07*, volume 5161 of *Lecture Notes in Computer Science*, pages 1–40, Cluj-Napoca, Romania, 2008. Springer-Verlag.

[PAK+08b] Rinus Plasmeijer, Peter Achten, Pieter Koopman, Bas Lijnse, and Thomas van Noort. An iTask case study: a conference management system. In *Revised Lectures of the 6th International Summer School on Advanced Functional Programming, AFP '08*, volume 5832 of *Lecture Notes in Computer Science*, Center Parcs "Het Heijderbos", The Netherlands, 19-24, May 2008. Springer-Verlag.

[PAK+10] Rinus Plasmeijer, Peter Achten, Pieter Koopman, Bas Lijnse, Thomas van Noort, and John van Groningen. iTasks for a Change: Enabling run-time change in an embedded workflow language. In *Submitted for the 15th International Conference on Functional Programming, ICFP '10*, Baltimore, Maryland, USA, 27-29, September 2010.

[PAP05] Rinus Plasmeijer, Peter Achten, and Javier Pomer Tendillo. iData for the world wide web - Generic programming techniques for high-level server-side web scripting. Internal report, Institute for Computing and Information Sciences, Radboud University Nijmegen, The Netherlands, February 2005.

[Pat] Workflow Patterns. Workflow Patterns Vendors. `http://www.workflowpatterns.com/vendors/`.

[Pau05] Linda Dailey Paulson. Building Rich Web Applications with Ajax. *Computer*, 38(10):14–17, 2005.

[PE01] Rinus Plasmeijer and Marko van Eekelen. *Concurrent Clean language report (version 2.0)*, December 2001. `http://clean.cs.ru.nl`.

[Pes08] Maja Pesic. *Constraint-based workflow management systems: shifting control to users.* PhD thesis, Technical University Eindhoven, 8, October 2008.

[Pil99] Marco Pil. Dynamic types and type dependent functions. In Kevin Hammond, Tony Davie, and Chris Clack, editors, *Proceedings of the 10th International Symposium on the Implementation of Functional Languages, IFL '98*, volume 1595 of *Lecture Notes in Computer Science*, pages 169–185, London, UK, 1999. Springer-Verlag.

[PJ87] S.L. Peyton Jones. *The Implementation of Functional Programming Languages*. International Series in Computer Science. Prentice-Hall, 1987.

[PJ92] S.L. Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, 1992.

[PJ03] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003.

[PJKA08] Rinus Plasmeijer, Jan Martin Jansen, Pieter Koopman, and Peter Achten. Declarative Ajax and client side evaluation of workflows using iTasks. In *Proceedings of the 10th International Conference on Principles and Practice of Declarative Programming, PPDP '08*, pages 56–66, Valencia, Spain, 15-17, July 2008.

[PJL92] Simon L. Peyton Jones and David R. Lester. *Implementing functional languages*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992. available at: `http://research.microsoft.com/en-us/um/people/simonpj/papers/pj-lester-book/`.

[PJW93] Simon Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the 20th International Symposium on Principles of Programming Languages, POPL '93*, pages 71–84, Charleston, SC, USA, January 1993. ACM Press.

[PLZ09] Hans Peukert, David Lincourt, and Birgit Zimmermann. Support for agile planning & execution of coordinated actions. In *Proceedings of 14th ICCRTS C2 and Agility*, 2009.

[PvE93] R. Plasmeijer and M. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. International Computer Science Series. Addison-Wesley, 1993.

[Rei98] Claus Reinke. Towards a Haskell/Java connection. In *Implementation of Functional Languages, IFL 1998*, volume 1595 of *Lecture Notes in Computer Science*, pages 203–219. Springer, 1998.

[RHAM06] Nick Russell, Arthur ter Hofstede, Wil van der Aalst, and Nataliya Mulyar. Workflow control-flow patterns: A revised view. Bpm center report bpm-06-22, BPM Center, 2006. `http://www.BPMcenter.org`.

[SAP] The Sapl Home Page. `http://home.kpn.nl/janmartinjansen/sapl`.

[SB09] Christian Sell and Iris Braun. Using a workflows management system to manage emergency plans. In J. Landgren and S. Jul, editors, *Proceedings*

of the 6th International ISCRAM Conference - Gothenburg, Sweden, May 2009.

[SGL06] Manuel Serrano, Erick Gallesio, and Florian Loitsch. Hop, a language for programming the web 2.0. In *Proceedings of the 11th International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '06*, pages 975–985, Portland, Oregon, USA, 22-26, October 2006.

[SM89] J. Steensgaard-Madsen. Typed representation of Objects by Functions. *ACM Transactions on Programming Languages and Systems*, 11(1):67–89, jan 1989.

[Sof] Software Technology Group, the Institute of Information and Computing Sciences, Utrecht University, the Netherlands. *The Helium Project.* `http://www.cs.uu.nl/helium`.

[Sol] BPS Solutions. COSA Workflow. `http://www.bps-solutions.de`.

[SP02] Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In Manuel M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, October 2002.

[Stu08] Aaron Stump. Directly reflective meta-programming. *Journal of Higher Order and Symbolic Computation*, 2008.

[Sun08] Sun Microsystems. Release notes for the next-generation Java Plug-In technology, 2008. `http://jdk6.dev.java.net/plugin2`, visited March 2009.

[Thi02] Peter Thiemann. WASH/CGI: server-side web scripting with sessions and typed, compositional forms. In Shriram Krishnamurthi and Raghu Ramakrishnan, editors, *Proceedings of the 4th International Symposium on the Practical Aspects of Declarative Programming, PADL '02*, volume 2257 of *LNCS*, pages 192–208, Portland, OR, USA, 19-20, January 2002. Springer-Verlag.

[TIB] TIBCO^{TM}. StaffWare. `http://www.staffware.com`.

[Tur79] D. A. Turner. A new implementation technique for applicative languages. *Softw., Pract. Exper.*, 9(1):31–49, 1979.

[VP03] Martijn Vervoort and Rinus Plasmeijer. Lazy dynamic input/output in the lazy functional language Clean. In Ricardo Peña and Thomas Arts, editors, *Revised Selected Papers of the 14th International Symposium on the Implementation of Functional Languages, IFL '02*, volume 2670 of *Lecture Notes in Computer Science*, pages 101–117, Madrid, Spain, September 2003. Springer-Verlag.

[VRA08] Irene Vanderfeesten, Hajo Reijers, and Wil van der Aalst. Product based workflow support: dynamic workflow execution. In Z. Bellahsène and M. Léonard, editors, *Proceedings of the 20th International Conference on Advanced Information Systems Engineering, CAiSE '08*, volume 5074 of *LNCS*, pages 571–574, Montpellier, France, 2008. Springer-Verlag.

[W3 08] W3 Schools. Ajax tutorial, 2008. `http://w3schools.com/ajax`.

[Wee07] Arjen van Weelden. *Putting types to good use*. PhD thesis, Institute for Computing and Information Sciences, Radboud University Nijmegen, The Netherlands, 17, October 2007. ISBN 978-90-9022041-3.

[YAW] YAWL Foudation. YAWL: Yet Another Workflow Language. `http://www.yawlfoundation.org`.

Summary

During the last decade the Internet has become the prominent platform for the deployment of computer applications. Nowadays, web-browsers are an important interface for a large class of computer applications, such as e-mail applications, online shops and banking applications and they are used as the default communication interface between customers and companies like governmental institutions, insurance companies, etc. An important advantage of using web-browsers as interface for applications is that they do not require installation of application related software on a computer to use them. It is even possible to run the same web application on a large number of different client platforms and operating systems, including PDA's, smart phones, etc. Despite this popularity and convenience for the user, for a software engineer the development of web application is a difficult job, because Internet applications are complex client-server applications that have a more complex structure than desktop applications.

For traditional Internet applications all processing is done at the server side and the Internet browser is only used to display information. Nowadays, Internet applications are much more interactive and require client-side processing for a better performance and an acceptable user experience. Several solutions exist for the realization of client-side processing. Many of them include the use of JavaScript at the client side. JavaScript is a programming language for which an interpreter is included in all popular web-browsers. But including an execution platform at the client side complicates the development of Internet applications considerably. The programmer now has to develop code both for the server and client side of the application and these parts should co-operate closely to obtain the desired result.

Functional programming languages like Haskell and Clean are a promising implementation platform for the development of web applications because of their high expressiveness. They support higher order combinators that enable a high level of compositional programming where irrelevant details can be hidden for the developer. They support generic programming techniques for automatic generation and handling of web forms, interaction with data sources and server-client communication. An important example of the functional programming approach to web development is the iTask system. The iTask system is a declarative domain specific language (DSL) embedded in Clean, enabling the creation of dynamic workflow systems. In the iTask system a workflow consists of a combination of **tasks** to be performed by humans and/or automated processes. From iTask specifications complete workflow applications are generated that run on the web. The iTask system is built on a single, powerful, concept: the task. The system uses combinators to

combine tasks into new tasks. With combinators tasks can be executed sequentially or in parallel using or- and- or ad-hoc parallelism. The original iTask system was a pure server side based application where all processing is done at the server side. The main object of study for this thesis is the extension of iTask with client-side processing while maintaining the highly declarative nature of the system and the generation of the complete application from one source in Clean.

This thesis consist of three parts. The first part discusses how client-side processing can be realized using a dedicated interpreter called Sapl (**S**imple **A**pplication **P**rogramming **L**anguage) that can be used as a web-browser plug-in. For this interpreter both the formalism of the interpreted programming language and the implementation of the interpreter are discussed. For the formalism a relatively unknown encoding of Algebraic Data Types by functions is used. It turns out that this encoding allows for an elegant and efficient implementation. It is also investigated whether the techniques used for the implementation of the interpreter can also be used for the realization of an efficient compiler.

The second part of this thesis discuses the addition of client-side processing to iTask. Two extensions are discussed. The first one adds partial updates of web pages using local task-tree rewriting and client-side task evaluation to the system. For client-side evaluation of tasks the previously mentioned interpreter is used. The implementation of these additions depends heavily on Clean Dynamics and its extension Clean-Sapl Dynamics. This makes it possible to serialize Clean functions, to maintain them for later execution, or to move them to the client to execute them there. The second extension of iTask enables the insertion of plug-ins into iTask applications and to exchange information with these plug-ins in a generic way. An extra feature of this integration is the possibility for a plug-in to use Clean functions as call-back mechanism for handling events. Events can be handled both on the server as well as on the client. In this way interactive iTask applications (iEditors) using plug-ins like graphical editors can be created. This extension of iTask uses the already mentioned Clean-Sapl Dynamics in combination with the possibility to define generic type driven functions in Clean.

The last part of this thesis discusses the possible application of iTask to the domains of Military and Crisis-Management Operations. These operations involve cooperation and collaboration between diverse organizations. Activities in these operations are highly dynamic and situation dependent. To cooperate and collaborate, activities by organizations must be synchronized (or at least de-conflicted) with one another. It is argued that properties of iTask like: data dependent workflows, exceptions, and the possibility to adapt workflows, already make iTask a good candidate to support these domains. But it is also argued that challenges like: better collaboration on tasks and offering a dedicated user interfaces to adapt workflows, should be further investigated to make iTask really useful for these areas.

Samenvatting

Het Internet is de laatste jaren uitgegroeid tot een belangrijk platform voor computer toepassingen. Tegenwoordig zijn web-browsers vaak de interface van computer applicaties. Ze worden gebruikt als interface voor e-mail, on-line winkelen en bankieren en als standaard communicatie kanaal tussen klanten en bedrijven zoals overheidsinstellingen, verzekeringsmaatschappijen, enz. Een belangrijk voordeel is dat er geen installatie van toepassingsgerelateerde software op de computer nodig is om ze te kunnen gebruiken. Het is zelfs mogelijk om dezelfde toepassing te gebruiken op een groot aantal verschillende platformen en besturingssystemen, waaronder PDA's, smart phones, enz. Ondanks deze populariteit en het gemak voor de gebruiker is het voor een software ontwikkelaar moeilijk om software voor het Internet te ontwikkelen, omdat Internet toepassingen gecompliceerde client-server toepassingen zijn die een ingewikkelder structuur dan desktop applicaties hebben.

Voor traditionele Internet applicaties wordt al het rekenwerk aan de kant van de server gedaan en wordt de Internet browser alleen gebruikt om informatie weer te geven. Tegenwoordig zijn Internet toepassingen veel interactiever en vereisen ze de mogelijkheid om aan de kant van de browser berekeningen uit te voeren teneinde een voor de gebruiker aanvaardbare efficiëntie te bereiken. Er bestaan hiervoor verschillende benaderingen. Velen zijn gebaseerd op het gebruik van JavaScript aan de kant van de browser. JavaScript is een programmeertaal waarvoor een interpreter is opgenomen in alle populaire web-browsers. Maar het toevoegen van processing aan de kant van de browser bemoeilijkt de ontwikkeling van Internet toepassingen aanzienlijk. De programmeur moet code ontwikkelen voor zowel server als client en deze delen moeten nauw samenwerken om het gewenste resultaat te verkrijgen.

Functionele programmeertalen zoals Haskell en Clean zijn een veelbelovend implementatie platform voor de ontwikkeling van web-applicaties vanwege hun hoge expressiviteit. Zij ondersteuenen hogere orde combinatoren die zorgen voor een hoog niveau van compositioneel programmeren waarbij irrelevante details verborgen blijven voor de ontwikkelaar. Zij ondersteunen het gebruik van generieke programmeertechnieken voor de automatische generatie en verwerking van webformulieren, interactie met gegevensbronnen en client-server communicatie. Een belangrijk voorbeeld van de functionele benadering is het iTask systeem. Het iTask systeem is een declaratieve domeinspecifieke taal (DSL) ingebed in de functionele programmeertaal Clean, voor het creëren van dynamische workflow systemen. In iTask bestaat een workflow uit een combinatie van taken die kunnen worden uitgevoerd door mensen en/of geautomatiseerde processen. Uit iTask specificaties worden complete workflow applicaties gegenereerd die worden uitgevoerd op het Internet. Het iTask systeem is

gebaseerd op een enkel krachtig concept: de taak. Het systeem gebruikt combinatoren om taken te combineren tot nieuwe taken. Met combinatoren kunnen taken sequentieel of (and-, or, ad-hoc) parallel worden uitgevoerd. Het oorspronkelijke iTask systeem was een pure server gebaseerde applicatie, waar alle processing op de server wordt gedaan. Het voornaamste doel van het onderzoek voor dit proefschrift is de uitbreiding van iTask met client-side processing met behoud van de declaratieve aard van het systeem en het genereren van een volledig systeem uit een enkel Clean programma.

Dit proefschrift bestaat uit drie delen. Het eerste deel behandelt hoe client-side processing kan worden gerealiseerd met behulp van de Sapl (**S**imple **A**pplication **P**rogramming **L**anguage) interpreter, die als plug-in in de web-browser wordt geladen. Van deze interpreter worden zowel het formalisme van de geïnterpreteerde programmeertaal als de implementatie van de interpreter zelf besproken. Voor het formalisme van de interpreter wordt een relatief onbekende codering van Algebraische Data Types met behulp van functies gebruikt. Het blijkt dat deze codering tevens een elegante en efficiënte implementatie toestaat. Het is tevens onderzocht of de technieken, gebruikt voor de implementatie van de interpreter, ook kunnen worden gebruikt voor de realisatie van een compiler.

Het tweede deel van dit proefschrift bespreekt de toevoeging van client-side processing aan iTask. Twee uitbreidingen worden besproken. De eerste voegt partiëel aanpassen van webpagina's en client-side taak evaluatie toe aan het systeem. Voor het laatste wordt de Sapl interpreter gebruikt. De implementatie van deze toevoegingen aan iTask is sterk afhankelijk van Clean Dynamics en de uitbreiding Clean-Sapl Dynamics. Hiermee is het mogelijk in een applicatie functies te serializeren en deze te bewaren voor latere uitvoering, of om ze te verplaatsen naar de client om ze daar uit te voeren. De tweede uitbreiding van iTask maakt de integratie van plug-ins in iTask applicaties mogelijk, alsmede de generieke uitwisseling van informatie met deze plug-ins. Een bijzonder kenmerk van deze integratie is de mogelijkheid voor een plug-in om Clean functies als call-back mechanisme te gebruiken voor de afhandeling van events. Events kunnen zowel op de server als op de client worden afgehandeld. Op deze manier kunnen interactieve iTask toepassingen (iEditors) met plug-ins zoals grafische editors worden gecreëerd. Deze uitbreiding van iTask maakt gebruik van de reeds genoemde Clean-Sapl Dynamics in combinatie met de mogelijkheid om generieke type gedreven functies in Clean te definiëren.

Het laatste deel van dit proefschrift bespreekt de mogelijke toepassing van iTask voor Militaire en Crisis Management Operaties. Deze operaties vereisen samenwerking van verschillende organisaties. Activiteiten in deze operaties zijn zeer dynamisch en situatie afhankelijk. Om goed te kunnen samenwerken moeten activiteiten van diverse organisaties worden gesynchroniseerd (of tenminste op elkaar worden afgestemd). Er wordt beargumenteerd dat eigenschappen van iTask zoals: data afhankelijke workflows, excepties en de mogelijkheid om workflows aan te passen, iTask een goede kandidaat maken om deze gebieden te ondersteunen. Maar er wordt ook beargumenteerd dat uitdagingen zoals: betere samenwerking bij taken en het aanbieden van specifieke user interfaces om taken aan te passen, verder onderzocht moeten worden om iTask echt geschikt te maken voor deze gebieden.

Dankwoord

Veel mensen hebben mij op een of andere wijze geholpen bij het tot stand komen van dit proefschrift. Met het noemen van namen loop ik het risico dat ik mensen vergeet en ik laat dit daarom ook maar achterwege. Er is echter een persoon die ik wel speciaal wil bedanken en dat is Dick Bruin. Ik kan wel zeggen dat ik zonder zijn inbreng nooit aan dit avontuur was begonnen. Zijn Amanda interpreter heeft mij lang geleden aan het functioneel programmeren gezet. De vele discussies die ik met hem heb gehad, hebben geleid tot het idee voor de codering van data structuren zoals gebruikt in de Sapl interpreter.

Verder wil ik collega's en kamergenoten van de Noordelijke Hogeschool Leeuwarden, CAMS Force Vision, de medewerkers van Software Technolgie en later MBSD van de Radboud Universiteit Nijmegen en mijn huidige collega's van de NLDA bedanken voor hun luisterend oor en stimulatie. Ook dank ik de deelnemers aan de Nederlandse FP dagen die mijn verhalen altijd welwillend hebben aangehoord en feedback hebben gegeven gedurende al weer bijna twee decenia.

Een speciaal woord van dank is er voor al mijn studenten gedurende al die jaren. Ik heb gemerkt dat bij vele van hen de functionele vonk is overgesprongen. Dat doet mij erg deugd!

Ik dank de leiding van CAMS Force Vision voor de ondersteuning die ik heb gehad in de eerste fase van mijn onderzoek en de leiding van de Nederlandse Defensie Academie voor de ondersteuning van het verdere traject.

Het thuisfront heeft de afgelopen jaren veel avonden en weekenden doorgemaakt, waarbij ik weliswaar fysiek aanwezig was, maar mentaal in functionele kringen verkeerde. Ik bedank Margriet, Mark en Marit voor het geduld dat ze hebben gehad.

Curriculum Vitae

1960	Born on March 14, Nieuwe Niedorp, the Netherlands
1972 - 1978	VWO, Alkmaar
1978 - 1983	Mathematics (cum laude), University of Amsterdam
1981 - 1982	Student assistent, University of Amsterdam
1982 - 1984	Student assistent, Center of Mathematics & Computer Science, Amsterdam
1984 - 1985	Scientific assistent, Free University, Amsterdam
1985 - 1991	Scientist, Philips Research, Eindhoven
1991 - 1999	Lecturer, NHL, Leeuwarden Polytechnic
1999 - 2006	System Engineer, Center for Automation of Mission Critical Systems, Ministry of Defense
2006 - now	Senior Lecturer, Netherlands Defense Academy, Ministry of Defense